图灵程序设计丛书

[PACKT] PUBLISHING

[美] David Herron 著　　鄢学鹍 吴天豪 廖健 译

Node Web Development

Node Web开发



人民邮电出版社
POSTS & TELECOM PRESS

David Herron

软件开发人员和软件质量工程师，在硅谷从业20多年，目前在雅虎担任质量工程小组的架构师，管控公司基于Node开发的Web应用平台的质量。

他曾为Sun公司主管工程师，并作为Java SE质量工程小组的架构师负责开发自动化测试工具（包括现在广泛用于GUI自动测试软件的AWT Robot类），期间参与了OpenJDK和JDK-Distros项目的启动，举办了世界性的Mustang Regressions大赛，让Java开发者社区寻找Java 1.6的bug。

任职Sun公司之前，他曾为VXtreme公司开发视频流处理工具（Windows Media Player的前身），在Wollongong集团从事电子邮件客户端和服务器软件的开发，加入了互联网工程任务组，负责改进与电子邮件相关的协议。

鄢学鹍

阿里云云手机开发者运营负责人，曾在网易做过UI设计师，在雅虎中国领导过前端团队，在口碑网领导过UED团队，还担任D2前端技术论坛顾问。对Web标准、前端开发模式、性能优化和自动化有较深入的研究。目前专注于从Mobile到PC领域的设计、技术和业务间的结合，常用ID：秦歌、三七。其译著有《JavaScript语言精粹》和《高性能网站建设进阶指南：Web开发者性能优化最佳实践》，个人博客是dancewithnet.com，Twitter账号是@kavenyan。

吴天豪

阿里云计算前端开发工程师，w3ctech杭州站负责人，w3ctech.com内容贡献者，负责过口碑网产品线的开发、基于移动浏览器的Web App开发，致力于构建快速、高效、可访问性高的Web应用。

廖健

阿里云资深前端开发工程师，有多年Flash平台开发经验，喜欢奔放的脚本语言，曾在D2论坛和HTML5研究小组线下沙龙做过技术分享，目前主要研究和实践Web技术在移动平台上的应用。

图书在版编目（C I P）数据

Node Web开发 / （美）赫伦（Herron,D.）著 ; 鄢学鹍，吴天豪，廖健译. -- 北京 : 人民邮电出版社，2012.4
（图灵程序设计丛书）
书名原文: Node Web Development
ISBN 978-7-115-27832-6

Ⅰ. ①N… Ⅱ. ①赫… ②鄢… ③吴… ④廖… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP393.092

中国版本图书馆CIP数据核字(2012)第056420号

内 容 提 要

Node 是一个服务器端的 JavaScript 解释器，是构建快速响应、高度可扩展网络应用的轻量高效的平台。Node 使用事件驱动和非阻塞的 I/O 模型，非常适合数据密集、对实时响应要求高的分布式应用。微软、eBay、LinkedIn、雅虎等世界知名公司及网站均有使用 Node 的成功案例。

本书是基于 Node 开发 Web 应用的实用指南，全书共分 6 章，通过示例详尽介绍了 Node 的背景、原理及应用方法。全书内容涉及 Node 简介、Node 安装、Node 模块、实现不同版本的简单应用、实现简单的 Web 服务器和 EventEmitter，以及数据存储和检索。另外，本书涵盖了 Node 服务器端开发的主要挑战及应对方案。

本书适合 Web 前、后端开发人员学习参考。

图灵程序设计丛书

Node Web开发

◆ 著　[美] David Herron
　译　鄢学鹍　吴天豪　廖　健
　责任编辑　毛倩倩

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷

◆ 开本：800×1000　1/16
　印张：7.25
　字数：176千字　　2012年 4 月第 1 版
　印数：1 − 5 000册　　2012年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-1950 号

ISBN 978-7-115-27832-6

定价：35.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

审稿人介绍

Blagovest Dachev　2002年起开始编写Web软件，从事过各种各样的开发工作，从HTML、CSS、JavaScript开发开始，后转站服务器和数据库领域。他很早就加入了Node.js开发行列，并已经为多个开源项目做出贡献，现在是Dow Jones &Company的一名软件工程师，致力于开发小工具框架，为第三方提供在自己网站上搜索和展示新闻的功能。

他曾就读于马萨诸塞大学安姆斯特分校,并在那儿参与了信息检索方面的研究,两次参与“谷歌编程之夏”（Google Summer of Code）活动，且与他人合著了几篇论文。

我想感谢我的母亲Tatiana，谢谢她给我的爱，对我的无私奉献，还有多年来一直激励我前进的力量；感谢父亲给了我一个幸福快乐的童年。

Matt Ranney　他很早就采用并参与开发Node.js，且是Voxer（使用Node作为后台服务器）的创建者之一。

致　谢

我想感谢很多人。

感谢母亲Evelyn为我做的一切，感谢父亲Jim、姐姐Patti、哥哥Ken。如果没有你们，真不敢想象我的生活会是个什么样子。

感谢女友Maggie一直以来对我的陪伴，给我的鼓励和信赖，她聪明睿智，总能在适当的时候为我打气。希望我们能永远在一起。

感谢肯塔基大学的Ken Kubota博士对我的信任，是他给了我第一份与计算机相关的工作。6年来，我不仅仅学到了计算机系统维护技巧，还学到了很多其他知识。

感谢我的前老板，感谢肯塔基大学数学科学系、Wollongong集团、MainSoft、VXtreme、Sun Microsystems和雅虎公司，还有每一位曾和我一起工作的同事。非常感谢我的前任经理Tina Su，是她不断鼓励我公开演讲和撰写文章，这对于一个内向的软件工程师而言绝非易事。非常感谢雅虎给我机会参与其内部的Node.js项目开发，并能够照顾我写作本书的需要。

感谢Packt Publishing给我写书的机会和给予的专业指导，让我认识到写书是我的梦想。

感谢Ryan Dahl、Isaac Schlueter和其他Node核心团队成员，是他们的智慧和眼光造就了如此有趣易用的软件开发平台。在诸多同类平台中， Node独树一帜，它既强大又好用，实现这么棒的平台需要极宽广的眼界。

www.packtpub.com

支持文件、电子书及打折优惠

欢迎读者访问www.packtpub.com下载支持文件及其他相关资源。

Packt为所有已出版图书提供电子版（PDF和ePub文件）。你可以在www.packtpub.com上升级电子书，且纸质书的用户有权享受购买其电子版的折扣。要了解更多信息，请发邮件到service@packtpub.com。

在www.packtpub.com还可以阅读免费的技术文章，通过注册收取一系列最新促销信息，并获得Packt纸质书及电子书的独家折扣或享受特价购书。



http://PacktLib.PacktPub.com

你是否经常受到一些IT难题的困扰？PacktLib是Packt的在线数字图书馆。在这里，你可以访问、阅读和搜索整个Packt图书馆里的图书。

为什么订阅

- 可以搜索Packt出版的所有图书；
- 可以复制、粘贴、打印内容并在内容上添加书签；
- 可以通过浏览器实现按需访问。

Packt 注册用户大礼

在www.packtpub.com上注册一个Packt账号，就可以立即访问PacktLib并自由浏览9本图书。注册→登录→浏览，就这么简单。

前　　言

欢迎光临Node（也叫Node.js）开发的世界。Node是一种新兴的软件开发平台，它将JavaScript从Web浏览器移植到常规的服务器端。Node运行在Chrome的高速V8引擎上，并附带了一个快速、健壮的异步网络I/O组件库。Node主要用于构建高性能、高可扩展的服务器和客户端应用，以实现真正"实时的Web应用"。

在经过数年尝试用Ruby和其他语言实现Web服务器组件之后，Ryan Dahl在2009年开发了Node平台。这个探索使他从使用传统的、基于线程的并发模型转向使用事件驱动的异步系统，因为后者更简单（多线程系统以难于开发著称），系统开销更低（与对每个连接维护一个线程相比），因而能提高相应的速度。Node旨在提供一个"创建可扩展网络服务器的简单方式"。这个设计受到了Event Machine（Ruby）和Twisted框架（Python）的影响，并和它们有些类似。

本书致力于讲述如何用Node构建Web应用。我们会在书中介绍快速学习Node时一些必需的重要概念。本书会教你编写真正的应用，剖析其工作原理，并讨论如何在程序中应用这些理念。我们需要安装Node和npm，学习安装和开发npm包及Node模块。此外，我们还会开发一些应用，度量长时间运行的计算在Node的事件循环中的响应能力，介绍将高负载的工作分派到多个服务器的方法，并介绍Express框架。

本书内容

第1章"Node入门"，介绍了Node平台。这一章讲述了Node的用途、技术构架上的选择、Node的历史和服务器端JavaScript的历史，然后介绍为什么JavaScript仍将受困于浏览器。

第2章"安装并配置Node"，介绍如何配置Node开发环境，包括多种从源码编译和安装的场景，还会简单介绍在开发环境中如何部署Node。

第3章"Node模块"，解释了作为开发Node应用基本单位的模块。我们会全面介绍并开发Node模块。然后进一步介绍Node包管理器npm，给出一些使用npm管理已安装包的例子，还将涉及开发npm包并将其发布出来供他人使用。

第4章"几种典型的简单应用"，在读者已经有一些Node基础知识后，开始探索Node应用的开发。我们会分别使用Node、Connect中间件框架和Express应用框架开发一个简单的应用。虽然应用比较简单，但是我们可以通过其开发探索Node的事件循环，处理长时间的运算，了解异步和同步算法以及如何将繁重的计算交给后台服务器。

第5章“简单的Web服务器、EventEmitter和HTTP客户端”，介绍了Node里的HTTP客户端和服务器对象。我们会在开发HTTP服务器和客户端应用的同时全面深入介绍HTTP会话。

第6章“存取数据”，探讨大部分应用都需要的长期可靠的数据存储机制。我们会用SQL和MongoDB数据库引擎实现一个应用。在此期间，我们将用Express框架实现用户验证，更好地展示出错页面。

阅读要求

目前，我们一般会采用源码的方式安装Node，这种方式可以很好地用在类Unix和符合POSIX标准的系统上。当然，在接触Node之前，谦逊的心态是必需的，但最为重要的事情还是让大脑供血充足。

从源码安装的方式需要一个类Unix或类POSIX系统（比如Linux、Mac、FreeBSD、OpenSolaris等）、新的C/C++编译器、OpenSSL库和Python 2.4或更新版本。

Node程序可以用任何文本编辑器来写，不过一个能处理JavaScript、HTML、CSS等的文本编辑器会更有帮助。

尽管本书介绍的是Web应用开发，但你并不需要拥有一个Web服务器。Node有自己的Web服务器套件。

读者对象

本书写给所有想在一个新的软件平台上开拓新编程模式的软件工程师。

服务器端程序员或许能看到一些新奇的概念，对Web应用开发产生新的理解。JavaScript是一门强大的语言，Node的异步特性发挥了JavaScript的优势。

浏览器端JavaScript“攻城师”或许会觉得在Node中使用JavaScript和编写与DOM操作无关的JavaScript代码很有趣。（Node平台上没有浏览器，所以也没有DOM，除非你安装JSDom。）

虽然本书各章内容由浅入深，循序渐进，但到底如何阅读本书悉听尊便。

本书需要读者知道如何编写软件，并且对JavaScript等编程语言有所了解。

排版约定

在本书中，读者会发现不同的文本样式。下面是这些样式的示例和说明。

正文中的代码使用特殊字体：“`http`对象封装HTTP协议，它的`http.createServer`方法会创建一个完整的Web服务器，而`.listen`方法用于监听特定的端口。”

代码块是这样的：

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
```

```
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

代码块中会加粗突出显示代码，这表示需要读者格外注意：

```
var util = require('util');
var A = "a different value A";
var B = "a different value B";
var m1 = require('./module1');
util.log('A='+A+' B='+B+' values='+util.inspect(m1.values()));
```

命令行的输入输出是这样的：

```
$ sudo /usr/sbin/update-rc.d node defaults
```

新术语及**重要词汇**都会加粗显示。你将在屏幕上看到的文字，比如菜单或对话框中的文字，会这样在正文中提到："一个真正安全的系统至少会有用户名和密码输入框。不过，我们这里就直接让用户单击Login按钮了。"

读者反馈

我们始终欢迎来自读者的反馈意见。我们想知道读者对本书的看法，读者喜欢哪些内容或不喜欢哪些内容。读者真正深有感触的反馈，对于我们开发图书产品至关重要。

一般的反馈可以发邮件到feedback@packtpub.com，但请在邮件标题中注明相关书名。

如果有关于新书的建议，你可以登录www.packtpub.com，填写SUGGEST A TITLE表单或者向suggest@packtpub.com发送邮件。

如果你在某个领域积累了丰富的经验，想写一本书，或者愿意与人合著或审校某本书，请阅读www.packtpub.com/authors上的作者指南。

读者服务

现在你已是Packt引以为荣的读者了，因此我们特别要交待几件事，以保障你作为读者的最大权益。

下载示例代码

在www.packtpub.com通过自己的账号购买图书的读者，可以下载所有已购买图书的代码[①]。如果这本书是你在其他地方购买的，访问www.packtpub.com/support并注册，我们将通过电子邮件将相关文件发送给你。

① 本书的代码文件也可在图灵社区（ituring. com. cn）本书网页免费注册下载。——编者注

勘误

虽然我们会全力确保本书内容的准确性,但错误仍在所难免。如果你发现了本书中的错误(包括文字和代码错误),而且愿意向我们提交这些错误,我们会十分感激。这样一来,不仅可以减少其他读者的疑虑,也有助于本书后续版本的改进。要提交错误,请访问www. packtpub.com/support,选择相关图书,单击errata submission form链接,然后输入勘误信息。经过验证后,你提交的勘误信息就会添加到已有的勘误列表中。要查看已有的勘误信息,请访问www.packtpub.com/support并选择相关图书。

反盗版声明

网上各种形式的盗版是一直存在的问题。Packt非常重视版权和许可证的保护。如果你在网上遇到以任何形式非法复制的我方作品,请尽快告知我们相关的地址或网站名称,以便我们采取补救措施。

请把邮件发送到copyright@packtpub.com,并在邮件里注明涉嫌侵权资料的链接。

感谢你帮助我们保护作者和我们为你带来有价值内容的能力。

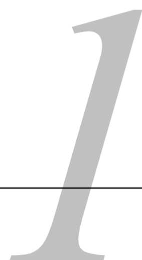
疑难解答

如果对本书的某些方面有疑问,请将电子邮件发送到questions@packtpub.com,我们会尽力解决。

目　录

第 1 章

Node入门



无论开发Web应用、应用服务器、任何类型的网络服务器或客户端还是通用编程，Node都是一个令人兴奋的新平台。它把异步I/O和服务器端JavaScript巧妙地组合在一起，聪明地运用了强大的JavaScript匿名函数和单线程执行的事件驱动架构，是专为网络应用的极高扩展性而设计的。

Node模型与大规模使用线程的普通应用服务器平台截然不同。由于使用事件驱动架构，内存占用量低，吞吐量高，且编程模型更简单。Node平台正处于高速成长阶段，很多人使用Node就是为了替代传统方法（使用Apache、PHP、Python等）。

Node的核心是一个独立的JavaScript虚拟机，通过扩展之后可适用于通用编程，并明确地专注于应用服务器开发。Node平台和开发Web应用的常用编程语言（PHP、Python、Ruby、Java等）不是一类东西，与那些向客户端提供HTTP协议的容器（Apache、Tomcat、Glassfish等）也没有直接可比性。而且，很多人认为它非常有可能取代传统的Web应用开发相关技术。

Node实现以非阻塞的I/O事件循环机制和文件与网络I/O库为中心，一切都以（来自Chrome浏览器的）V8 JavaScript引擎为基础。这个I/O库足以实现采用任何TCP或UDP协议的、任意类型的服务器，无论是DNS服务器，还是采用HTTP、IRC、FTP服务器。虽然它支持针对任何网络协议开发服务器或客户端，但最常用于构建普通网站，这种情况下可以取代像Apache/PHP或Rails这样的框架。

本书将向你介绍Node。我们假定你已知道如何编写软件、熟悉JavaScript，且对如何用其他语言开发Web应用有所了解。我们将以开发实际应用为例，因为阅读实际的代码是最好的学习方式。

1.1 Node 能做什么

Node是脱离浏览器编写JavaScript应用的平台。这里的JavaScript并非我们在浏览器中熟悉的JavaScript，Node没有内置DOM，也没有任何浏览器的功能。但它使用JavaScript语言和异步I/O框架，是一个强大的应用开发平台。

Node无法用于实现桌面GUI应用。目前Node既没有内置相当于Swing（或SWT）的功能组

件[①]，也没有Node扩展GUI工具包，而且不能内嵌到Web浏览器中。如果有可用的GUI工具包，Node就能用于构建桌面应用。一些项目已经开始为Node创建GTK绑定[②]，它能提供一个跨平台的GUI工具包。Node使用的V8引擎带有一个扩展API，允许编入C/C++代码以扩展JavaScript或集成原生代码库。

除了能原生执行JavaScript外，Node捆绑的模块还能提供如下功能：

- 命令行工具（shell脚本风格）；
- 交互式TTY风格编程（REPL[③]）；
- 出色的进程控制函数能监控子进程；
- 用Buffer对象处理二进制数据；
- 使用全面事件驱动回调函数的TCP或UDP套接字；
- DNS查找；
- 基于TCP库的HTTP和HTTPS客户端/服务器；
- 文件系统的存取；
- 内置了基于断言的单元测试能力。

Node的网络层是底层，但易于使用。例如，HTTP模块可以让你仅用几行代码编写出一个HTTP服务器（客户端），但这层让程序员非常贴近协议的要求，且让你精确控制将返回的请求响应的HTTP头。PHP程序员通常不需要关心HTTP头，但Node程序员需要。

换句话说，用Node编写HTTP服务器非常容易，但通常Web应用开发者不需要在这种细节层面上费神。例如，由于Apache已经存在，PHP程序员就不需要实现其HTTP服务器部分了。Node社区已经开发出许多类似于Connect的Web应用框架，让开发者能够快速配置可提供所有基础内容（会话、cookie、静态文件和日志等）的HTTP服务器，从而把时间和精力都放在业务逻辑上。

服务器端 JavaScript

有点不耐烦了？你正一边摇头一边抱怨："在服务器上用一门浏览器语言做什么？"实际上，JavaScript在浏览器之外有着悠久且大量鲜为人知的历史。JavaScript就像其他语言一样是一门编程语言，而你的问题可能应该这样："为什么JavaScript会一直被困在浏览器中?"

---

① Swing是一个为Java设计的工具包，是Java基础类的一部分，具体见http://zh.wikipedia.org/wiki/Swing_(Java)。SWT（Standard Widget Toolkit）最初由IBM开发，是一套用于Java的GUI系统，用来和Swing竞争，而开源集成开发环境Eclipse就是用Java和SWT开发的，具体见http://en.wikipedia.org/wiki/Standard_Widget_Toolkit。（本书脚注若无特殊说明均为译者注。）

② GTK+使用C语言开发，是类Unix系统下开发GUI应用程序的主流开发工具之一。它是自由软件，并是GNU计划的一部分，具体见http://zh.wikipedia.org/wiki/GTK。

③ REPL就是Read-Eval-Print Loop的缩写，意思是"阅读–评估–打印–循环"，它既可以作为独立的程序运行，也很容易作为程序的一部分使用。它为运行JavaScript脚本和查看结果提供了一种交互方式，详细内容见http://nodejs.org/docs/v0.6.6/api/repl.html 和http://nodejs.org/docs/v0.6.6/api/tty.html。

Web诞生之初，编写Web应用的工具还不够成熟。有些人尝试用Perl或TCL编写CGI脚本，PHP和Java语言刚刚被开发出来，而JavaScript甚至也被用于服务器端。Netscape的LiveWire服务端作为早期的Web应用服务器，就是用JavaScript编写的。微软ASP的某些版本使用的是JScript——它们自己的JavaScript实现。最近的一个服务器端JavaScript项目是在Java领域中使用的RingoJS应用框架。RingoJS构建于Rhino[①]之上，Rhino是一个用Java编写的JavaScript实现。

Node带来了一个前所未见的组合，那就是高速事件驱动I/O和V8高速JavaScript引擎；V8这个超快的JavaScript引擎是Google Chrome浏览器的核心。

1.2　为什么要使用 Node

JavaScript由于无处不在的浏览器而非常流行。它实现了许多现代高级语言的概念，比其他任何语言都不逊色。多亏了它的普及，软件行业才有大量经验丰富的JavaScript人才储备。

JavaScript是一门动态编程语言，拥有松散类型且可动态扩展的对象（能按需非正式地声明）。函数是一级对象，通常作为匿名闭包使用。这使得JavaScript比其他常用于编写Web应用的语言更加强大。理论上，这些特性使开发者的工作更加高效。平心而论，动态和非动态、静态类型和松散类型语言之间的优劣尚无定论，而且可能永远不会有定论。

JavaScript的一个短板是全局对象。所有的顶级变量都被扔给一个全局对象，这在混用多个模块时会导致难以预料的混乱。由于Web应用通常有大量的对象，且很可能是多个组织编写的，所以你自然会认为Node编程中的全局对象冲突会是个“雷区”。但其实不然，Node使用CommonJS[②]模块系统，这意味着模块的局部变量即使看起来像全局变量，实际上也是局部变量。这种模块间的清晰分离避免了全局对象的问题。

在Web应用服务器端和客户端使用同样的编程语言是人们由来已久的梦想。这个梦想可以追溯到早期的Java时代，那时Applet是用Java编写的服务器应用的前端，而对JavaScript的最初设想是将其作为Applet的一种轻量级脚本语言。但世事难料，到头来JavaScript取代Java成为在浏览器中使用的唯一语言。有了Node，在客户端和服务器端使用相同编程语言的梦想终于有望实现了，这门语言就是JavaScript。

语言在前后端通用有如下几个优势：

- 网线两端可能是相同的程序员；
- 代码能更容易地在服务器端和客户端间迁移；

① Rhino是用Java编写的JavaScript引擎，其设计目标是借助于强大的Java平台API简化JavaScript程序的编写。Rhino是Mozilla开发的自由软件，其最新的1.7r3版本实现了部分ECMAScript 5，可以从http://www.mozilla.org/rhino/下载它的源代码。

② CommonJS是一种规范，Node实现了这个规范。JavaScript是一门强大的、面向对象的语言，它有很多快速高效的解释器。JavaScript标准定义的API是为了构建基于浏览器的应用程序，不存在用于更广泛应用程序的标准库。CommonJS定义了构建很多普通应用程序（主要指非浏览器应用）的API，从而填补了这个空白。CommonJS的终极目标是提供一个类Python、Ruby和Java的标准库。这样的话，开发者可以使用CommonJS API编写应用程序，然后这些应用可以运行在不同的JavaScript解释器和不同的主机环境中。更多内容见http://www.commonjs.org/。

- 服务器端和客户端使用相同的数据格式（JSON）；
- 服务器端和客户端使用相同的开发工具；
- 服务器端和客户端使用相同的测试或质量评估工具；
- 当编写Web应用时，视图模板能在两端共享；
- 服务器端和客户端团队可使用相似的编程风格。

Node作为一个引人注目的平台，加上开发社区的共同努力，很容易把上述这些（甚至更多）优势变成现实。

1.2.1　架构问题：线程，还是异步事件驱动

Node的异步事件驱动架构据说是其拥有极高性能的原因。好吧，应该说还有V8 JavaScript引擎的巨大功劳。通常应用服务器端使用阻塞I/O和线程来实现并发。阻塞I/O会导致线程等待，从而造成线程资源浪费，因为当应用服务器处理请求时，需要等待I/O执行结束才能继续处理。

Node有一个无需I/O等待或执行环境切换的单独执行环境。Node的I/O调用会转换为请求处理函数，当某些数据可用时事件轮询会调度事件让这些函数工作。事件轮询和事件处理程序模型差不多，就像浏览器中的JavaScript一样。程序的执行最好能快速返回到事件循环中，以便马上调度下一个可执行的任务。

为了说明这一点，Ryan Dahl（在他的“Cinco de Node”演讲[①]中）问我们当执行如下代码时会发生什么：

```
result = query('SELECT * from db');
```

当然，在数据库层把这个查询发送到数据库、数据库查询结果并返回数据期间，程序会暂停。根据具体的查询情况，暂停时间可能相当长。这非常糟糕，因为线程空闲时可以处理另一个请求，如果所有线程都繁忙（记住计算机资源有限），请求将被丢弃。这当然是浪费资源。执行环境切换也不是没有代价的，使用的线程越多，CPU存储和恢复状态消耗的时间就越多。此外，每个线程的执行栈都占用内存。而是通过使用异步事件驱动的I/O，Node会省掉这里的大部分开销而其自己带来的开销却很小。

用线程实现并发通常面临类似这样的声音，即“开销大易出错”、“Java易出错的同步原语”或“设计并发软件复杂易出错”（实际上引号中的内容来自真实的搜索引擎结果）。复杂性来自于对共享变量的访问、避免死锁的各种策略和线程间的竞争。“Java的同步原语”就是这样一种策略，显然许多程序员发现其难以实现，因此倾向创建类似`java.util.concurrent`这样的框架来解决线程并发的问题，但有些人可能会认为掩盖复杂性并不会使事情更简单。

Node从不同的角度解决并发问题。通过事件轮询实现异步触发回调函数是一种更简单的并发模型，好理解且好实现。

Ryan Dahl指出理解读取对象的时间可以帮助理解异步I/O的重要性。从内存中读取的对象（纳

① 这是雅虎主办的技术会议，在这个会议上Ryan Dahl介绍了Node。具体情况和视频见http://www.yuiblog.com/blog/2010/05/20/video-dahl/。

秒级）比从硬盘或网络中读取对象（毫秒级或秒级）要快很多。读取外部对象的时间非常长，当用户等待页面加载完成，坐在浏览器前无聊地移动鼠标超过两秒时，这就显得没完没了了。

在Node中，前面的查询应该这样写：

```
query('SELECT * from db', function (result) {
  // 操作结果
});
```

这些代码实现了前面提到的查询。不同的是，查询结果不是调用函数的结果，而是提供给了一个稍后将调用的回调函数。此时，控制会立即返回事件循环，而服务器能够继续处理其他请求。这些请求中将会有一个是对查询的响应，那么该请求将调用回调函数。这种快速返回事件循环的模型确保了更高的服务器利用率。对于服务器的拥有者来说这太棒了，但更大的好处是它能让用户更快看到页面内容。

通常网页会引用几十个来源的数据，每个请求都会涉及上面讨论的查询和响应。通过异步查询，它们能并行发生，所以页面构造函数能同时发出几十个查询——无需等待且每个都有自己的回调函数——然后返回事件循环，而每个查询返回后会调用相应的回调函数。因为并行，所以收集数据比这些查询一个一个地同步完成要快得多。现在用户更高兴了，因为网页打开得更快了。

1.2.2　性能和利用率

Node之所以令人兴奋，还因为它的吞吐能力（每秒能响应的请求数）。与相似的应用（比如Apache）进行基准测试比较，可以看到Node的性能提升很大。

下面这个简单的HTTP服务器就是一个基准测试程序，它只是返回“Hello World”信息：

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

这是用Node能够构建的一个简单的Web服务器。`http`对象封装了HTTP协议，其`http.createServer`方法能创建一个完整的Web服务器，同时通过`.listen`方法监听指定端口。Web服务器上的每个请求（来自是任何URL的GET或PUT）都会调用这里提供的函数，就这么简简单单的几行代码。在这种情况下，无论是什么URL，它都会返回一个简单的`text/plain`响应“Hello World”。

由于简单，所以这个应用用以用来度量Node请求的最大值吞吐量。实际上许多已公布的基准测试程序都是从这个最简单的HTTP服务器开始的。

Node作者Ryan Dahl演示过一个简单的基准测试程序（http://nodejs. org/cinco_de_node.pdf），该程序返回一个1 MB的缓冲区（buffer），Node每秒能处理822个请求，而nginx每秒处理708个。他还指出nginx的峰值是4 MB内存，而Node是64 MB。

Dustin McQuay（http://www.synchrosinteractive.com/blog/9-nodejs/22- nodejs-has-a-bright-future）演示了据他所称类似的Node和PHP/Apache程序：

- PHP/Apache吞吐量为3187请求/秒；
- Node.js吞吐量为5569请求/秒。

RingoJS的作者Hannes Wallnöfer写了一篇博文提醒大家不要根据基准测试结果做重要决策（http://hns.github. com/2010/09/21/benchmark.html），随后使用基准测试程序比较了RingoJS和Node。RingoJS是一个应用服务器，构建在基于Java的Rhino JavaScript引擎之上。取决于应用场景，RingoJS和Node的性能没有多大区别。测试结果表明对于具有快速缓冲区或字符串分配机制的应用，Node的性能不如RingoJS。在后来的一篇博文中，他使用解析JSON字符串来模拟一个常见的任务，并发现RingoJS表现得更好。

Mikito Takada发表了关于基准测试和性能改进的博文，文中比较了基于Node构建的“48 hour hackathon”（48小时黑客马拉松）应用（http://blog.mixu.net/2011/01/17/performance-benchmarking-the-node-js-backend-of-our-48h-product-wehearvoices-net/）和一个他声称使用Django编写的类似应用。未经优化的Node版本相比Django版本有点儿慢（响应时间长），但一些优化（MySQL连接池、缓存等）极大地改进了其性能，使其轻而易举的击败了Django。最终的性能图表显示其性能（请求数/秒）几乎可与之前讨论的简单“Hello World”基准测试程序相媲美。

注意，Node应用高性能的关键是要快速返回事件循环。我们会在第4章对此进行详细介绍，如果回调处理程序执行时间“太长”，用Node也很难成就极快的服务器。在Ryan Dahl关于Node项目的一篇早期博文（http://four.livejournal.com/963421.html）中，他论述了事件处理程序执行时间要小于5 ms的必要性。这篇博文中的大部分想法都没有实现，但Alex Payne写了一篇关于它的有趣博文（http://al3x.net/2010/07/27/node.html）分析了“小规模应用”（scaling in the small）和“大规模应用”（scaling in the large）的区别。

对于小型Web应用，在用Node编码取代通常的“P”语言（Perl、PHP、Python等）编码时具有性能和实现上的优势。JavaScript是一门强大的语言，同时使用现代高速虚拟机设计的Node环比像PHP这样的解释语言更具性能和并发上的优势。

在讨论“大规模应用”时他说，企业级应用的编写总是困难且复杂的。典型的企业级应用都会使用负载均衡、缓存服务器、大量冗余机器，这些机器很可能分散在各个不同的地方，为世界各地的用户提供快速的Web浏览体验。或许对于整个系统来说应用开发平台并不那么重要。

在看到Node被真正长期实际地采用之前，很难知道它真正的价值有多大。

1.2.3　服务器利用率、成本和绿色 Web 托管服务

追求最佳效率（每秒内处理更多的请求）不仅仅是要通过优化得到极客一般的满足感，还要追求真正的商业和环境效益。像Node服务器那样，每秒处理更多的请求就意味着不必再购买大量的服务器。本质就是用更少的资源做更多的事情。

大致来说，购买更多的服务器，就要消耗更多电力和造成更大的环境污染，反之，服务器越

少，投入越少，对环境污染也越少。对于降低运行Web服务器成本和对环境的影响，现已存在一个专业的研究领域。这里的目标相当明显：更少的服务器、更低的成本和更小的环境污染。

英特尔的文章“通过服务器电能测量提高数据中心效率”（Increasing Data Center Efficiency with Server Power Measurements，http://download.intel.com/it/pdf/Server_Power_Measurement_final.pdf）给出了用于理解效率和数据中心成本的客观框架。有许多因素（诸如建筑、冷却系统和计算系统设计）都会对成本和环境产生影响。高效的建筑设计、冷却系统和计算机系统（数据中心效率、密度和存储密度）能降低成本和环境影响。但你可能因为部署一个低效的软件框架需要购买更多的服务器，以致毁掉这些收益，然而你也可以通过使用高效的软件框架放大数据中心的效率。

1.3 Node、Node.js 还是 Node .JS

这一平台的名字是Node.js，但本书使用Node这个拼写形式，因为我们遵循了nodejs.org网站的提示，它说Node.js（小写的.js）是商标，但整站都使用Node这个拼写形式，本书也与官方网站保持一致。

1.4 小结

本章介绍了很多知识，具体包括：

- JavaScript在浏览器之外亦有一番天地；
- 异步和阻塞I/O的不同之处；
- 关于Node的基本情况；
- Node的性能。

介绍完Node之后，我们准备深入讲述如何使用它。第2章将讨论如何设置Node环境，准备好了吗？

第 2 章

安装并配置Node

开始使用Node之前，需要先配置好开发环境。后面的几章里我们都会基于这个环境进行开发，不过这个配置不适用于生产环境的部署。

本章内容

- 了解如何利用源代码在Linux和Mac上安装Node；
- 了解如何安装npm包管理器和一些常用的工具；
- 学习一些关于Node模块系统的知识。

开始吧。

2.1　系统要求

Node最适合在符合POSIX[①]标准的操作系统上运行这些系统包括了UNIX的衍生系统（Solaris等）和通用计算机操作系统（Linux-Mac OS X等）。实际上，Node内置的很多函数都是直接调用POSIX系统的接口。

一些高级语言平台（比如Perl和Python）拥有稳定的特性和API，通常会被打包进不同版本的操作系统。由于Node还在快速发展中，将其打包进操作系统可能还为时尚早。这也意味着，Node还只能利用源代码安装。

利用源代码安装Node需要准备一个C语言编译器（比如GCC），还需要Python 2.4（或者更新版本）。如果打算对网络操作加密，你还需要OpenSSL加密库。现代的UNIX衍生系统大部分都自带了这些程序，Node的配置脚本（在下载并配置源文件的时候可以看到）会检测这些程序是否存在。如果你需要自己进行安装，可以在http://python.org上下载安装Python，而可以在http://openssl.org中找到并下载安装OpenSSL。

虽然Windows没有兼容POSIX标准，但我们能通过在Windows上模拟POSIX兼容环境来安装Node（Node 0.4.*x*或者更早的版本）。在0.6.*x*或者更新的版本中，Node开发团队打算将Node开发

---

① POSIX是IEEE开发的一系列互相关联的标准的总称，旨在为要在各种UNIX操作系统上运行的软件定义API，参见http://en.wikipedia.org/wiki/POSIX。

成可以在Windows中直接安装的程序。因为安装方式变化太快，所以本书不会介绍如何在Windows里安装Node，最新的安装说明见https://github.com/ry/node/wiki/Installation，其中的Step 3b部分讨论了如何通过使用Cygwin或者MinGW在Windows上安装Node。安装好了Cygwin或者MinGW，再安装Node的步骤与在符合POSIX标准的系统上是类似的。

2.2 在符合 POSIX 标准的系统上安装

现在你应该已经对安装Node大致有些了解，接下来让我们亲自体验下安装过程。一般情况下，安装需要执行`configure`、`make`、`make install routine`命令，而你应该已经在安装其他软件的时候尝试过这些操作了。

官方的安装说明请参考Node的wiki系统，地址是：

https://github.com/ry/node/wiki/Installation

安装条件

前面我们提到，安装Node需要准备3个程序，C语言编译器、Python和OpenSSL库。Node安装过程会先检测这些程序是否已经安装，如果C语言编译器或者Python缺失，安装就会失败。具体安装这些程序的方法因计算机操作系统而异。

下面的命令可以检查这些软件的安装情况：

```
$ cc --version
i686-apple-darwin10-gcc-4.2.1 (GCC) 4.2.1 (Apple Inc. build 5666) (dot 3)
Copyright (C) 2007 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
$ python
Python 2.6.6 (r266:84292, Feb 15 2011, 01:35:25)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.3 在 Mac OS X 上安装开发者工具

开发者工具（比如GCC）在Mac OS X上的安装是可选的，有两种安装方案，并且都是免费的。安装OS X的DVD中，有一个目录是“Optional Installs”（可选的安装程序），这个文件夹里包含了一个软件包安装程序，用于安装开发者工具，包括Xcode。

另外一个方案是从http://developer.apple.com/xcode/上下载最新版本的Xcode。

2.3.1 在 home 目录下安装

以前，将Node安装到home目录下是一个很好的方式，可方便以后的应用开发。不过现在，Node 0.4.*x*或者更新的版本带来了一些变化，特别是npm 1.0发布后，我们已经没有太大必要将

Node安装到home目录下。你可以将Node安装到系统级目录（见2.3.2节），或者当前用户目录下，从而方便开发或测试。

现在让我们看一看如何在当前用户目录下安装Node。

(1) 首先，从http://nodejs.org/#download上下载源代码。你可以通过浏览器下载，或者用下面的方式：

```
$ mkdir src
$ cd src
$ wget http://nodejs.org/dist/node-v0.4.8.tar.gz
$ tar xvfz node-v0.4.8.tar.gz
$ cd node-v0.4.8
```

(2)下一步是配置源代码以便其可以用于构建。其中包括一些典型的配置脚本，运行`./configure -help`可以看到巨多的选项。运行以下命令可以将Node安装到home目录下：

```
$ ./configure --prefix=$HOME/node/0.4.8
Checking for program g++ or c++          : /usr/bin/g++
Checking for program cpp                 : /usr/bin/cpp
Checking for program ar                  : /usr/bin/ar
Checking for program ranlib              : /usr/bin/ranlib
...
```

一段时间后，安装过程会停止，此时配置过程大概已经完成，源文件已经可以在你选择的目录下安装了。如果没有成功的话，控制台会打印出一条消息，告诉你需要解决的问题。如果配置脚本执行完毕，你就可以进行下一步了。

(3) 配置脚本执行完毕，就可以对Node进行编译了：

```
$ make
此处是长长的编译日志
$ make install
```

(4) 安装完成后，你需要将安装目录添加到`PATH`变量中：

```
$ echo 'export PATH=$HOME/node/0.4.8/bin:${PATH}' >>~/.bashrc
$ . ~/.bashrc
```

如果你使用C shell[①]，要使用的命令如下：

```
$ echo 'setenv PATH $HOME/node/0.4.8/bin:${PATH}' >>~/.cshrc
$ source ~/.cshrc
```

完成后会生成一些新目录：

```
$ ls ~/node/0.4.8/
bin     include     lib     share
$ ls ~/node/0.4.8/bin
node        node-waf
```

完成这一步，你就可以直接阅读2.4节了。

① C shell是Unix shell的一种，由Bill Joy在BSD系统上开发，参见http://zh.wikipedia.org/wiki/C_Shell。

为什么要安装到home目录

将Node安装到home目录主要考虑了两个原因：

- 方便测试和开发；
- 安全性。

首先，开发者如果考虑试验自己定制的Node实例，在多个版本的Node下测试自己的应用，甚至直接修改Node的源代码，我们推荐将Node安装到home目录下。

安全问题可能不太好理解，让我们慢慢分析。

首先，假设在使用类Unix系统时，你没有管理员权限，但是想要使用Node。如果你是在home目录下安装Node的话，这还是很方便的。

其次，在安装Node或相关工具（如npm）时下载和执行脚本也有风险。你能确保这些工具的作者足够可信吗？0.1.*x*或者0.2.*x*这样的版本号就说明不够稳定或不够安全。无论如何，通过`sudo`下载旧版本的npm会带来一些麻烦，而这就足以说明问题了。

在npm 1.0之前，所有的模块都必须安装在Node实例里。这看起来无伤大雅，但是当Node被安装在系统级目录下时则不然，这时操作需要root权限，并且在安装Node包时会执行很多脚本。你可能没有root权限，本地安全策略也可能会禁止一些乱七八糟的软件以root权限执行。如果在home目录下安装Node，任何破坏性的行为都会被限制在home目录下。祝你好运。

在Node 0.4.*x*和npm 1.0.*x*环境下，一般推荐将Node包安装到当前用户目录下，而不是在Node实例中安装。这样的安装过程是不需要root权限的。

正因为如此，我们现在可以在系统级目录下以管理员身份操作Node实例。因为Node拥有一个灵活的包查找算法，所以你也可以在当前用户目录上安装应用需要的Node包。我们会在下一章详细讲述这些内容。

2.3.2　在系统级目录下安装 Node

一般情况下，我们推荐你将Node安装到系统级目录下。原因有：

- 这是一个普遍的最佳实践；
- 这样Node就可以被多个不同应用或者用户共享；
- 在安装Node的时候可以避免不小心覆盖其他文件的问题；
- 允许你在系统启动的时候启动Node服务器。

在系统级目录下安装Node和在home目录下安装的过程基本一致，不过有两点不同之处。

- 第一点是选择安装目录的时候。我们用配置脚本来选择安装目录，默认情况下会安装到`usr/local`目录下：

  ```
  $ ./configure          # for /usr/local
  $ ./configure –prefix=/usr/local/node/0.4.8
  ```

  基本上，你只需要选择目录并使用`configure`命令完成配置。
- 第二点不同是`make install`步骤。由于系统级目录大都不允许普通用户写入文件，因此你需要取得root权限并用下面的方式安装：

```
$ sudo make install
```

注意，如果Node的安装目录已经在PATH环境变量中，就不需要重复修改了。

2.3.3　在 Mac OS X 上使用 MacPorts 安装

使用之前的方式，你也可以将Node安装到Mac OS X系统上。由于Mac OS X兼容UNIX，因此这是完全没有问题的。

MacPorts[①]项目（http://www.macports.org/）长久以来一直为Mac OS X系统提供一系列开源软件包，并且打包了Node。在你使用其网站上的安装器安装了MacPosts之后，安装Node就是非常简单的事了。

```
$ sudo port search nodejs
nodejs @0.4.8 (devel, net)
    Evented I/O for V8 JavaScript
$ sudo port install nodejs
下载并安装先决程序及Node的长长的日志
```

不过，不能通过这样的方式安装npm。

2.3.4　在 Mac OS X 上使用 homebrew 安装

homebrew是另外一个Mac OS X上的开源软件包管理工具，有些人甚至认为它是MacPorts的完美替代者。我们可以通过网站http://mxcl.github.com/homebrew/下载安装homebrew。按照网站上的说明安装完成后，可以按照下面的方式非常方便地安装Node：

```
$ brew search node
leafnode  node
$ brew install node
==> Downloading http://nodejs.org/dist/node-v0.4.8.tar.gz
######################################################### 100.0%
==> ./configure --prefix=/usr/local/Cellar/node/0.4.8
==> make install
.. etc
$ brew search npm
npm can be installed thusly by following the instructions at
  http://npmjs.org/
```

npm也可以通过上面这样的方式安装，具体请参照http://npmjs.org/上的说明。

2.3.5　在 Linux 上使用软件包管理系统安装

虽然Linux或者其他操作系统上还不会预装Node，但这并不意味着不能通过软件包管理器安装Node。Node wiki上的说明部分最近就列出了为Debian、Ubuntu、OpenSUSE和Arch Linux系统打包好的Node包。

① MacPorts曾经叫做 DarwinPorts[1]，是一个软件包管理系统，用来简化Mac OS X和Darwin操作系统上软件的安装。

读者可以通过访问https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager查看详细内容。

在Debian下的安装方法如下：

```
# echo deb http://ftp.us.debian.org/debian/ sid main >
/etc/apt/sources.list.d/sid.list
# apt-get update
# apt-get install nodejs # Documentation is great.
```

在Ubuntu下的安装方法如下：

```
# sudo apt-get install python-software-properties
# sudo add-apt-repository ppa:jerome-etienne/neoip
# sudo apt-get update
# sudo apt-get install nodejs
```

期待有一天Linux和其他操作系统能将Node以类似语言包的形式捆绑到OS中。

2.3.6　同时安装并维护多个 Node

通常你不会同时安装Node的多个版本，这样会让系统变得复杂。但是如果在开发Node，或者测试一些不同的Node版本，或者处于一些其他类似的情况下，你可能会安装多个Node。此时，我们只需对之前的步骤稍作修改，即可安装多个Node。

之前的安装说明部分曾提到过，-prefix选项可以用于在同一目录下并行安装多个不同版本的Node：

```
$ ./configure --prefix=$HOME/node/0.4.8
```

也可以这样安装：

```
$ ./configure --prefix=/usr/local/node/0.4.8
```

这一步决定了Node的安装目录。无论版本是0.4.9、0.6.1或者其他版本，你依然可以通过修改安装前缀同时安装多个版本。

Node版本之间的切换可以通过修改PATH环境变量实现（在符合POSIX标准的系统上），具体如下：

```
$ export PATH=/usr/local/node/0.6.1/bin:${PATH}
```

维护这么多版本有点麻烦。针对不同的版本，你需要启动Node，npm和其他安装在Node上的第三方模块。而修改PATH环境变量并不是理想的方式。别出心裁的程序员已经构建了多版本的管理器，来自动配置Node和npm并从而简化了操作。这个管理器也提供了一些命令用于更智能地修改PATH环境变量：

- https://github.com/visionmedia/n —— Node版本管理器；
- https://github.com/kuno/neco —— Nodejs Ecosystem COordinator（用于Node环境的协调）。

2.4　验证安装成功与否

现在已经安装好Node了，我们需要做两件事：验证安装是否成功，熟悉命令行工具。

2.4.1　Node 命令行工具

用普通方式安装的Node含有两个命令——`node`和`node-waf`。我们已经在前面看到过`node`。它可以用于运行命令行脚本或者服务器进程。`node-waf`命令是一个构建工具，用于创建Node原生的扩展（这部分内容不是本书关心的，所以我们不会在书中提及，你可以通过阅读nodejs.org上的在线文档了解更多信息）。

有一个最简单的方式来检查Node是否安装成功，同时它也是向Node获取帮助的最好方式。输入下面的命令：

```
$ node --help
Usage: node [options] script.js [arguments]
Options:
  -v, --version        print node's version
  --debug[=port]       enable remote debugging via given TCP port
                       without stopping the execution
  --debug-brk[=port]   as above, but break in script.js and
                       wait for remote debugger to connect
  --v8-options         print v8 command line options
  --vars               print various compiled-in variables
  --max-stack-size=val set max v8 stack size (bytes)

Enviromental variables:
NODE_PATH              ':'-separated list of directories
                       prefixed to the module search path,
                       require.paths.
NODE_DEBUG             Print additional debugging output.
NODE_MODULE_CONTEXTS   Set to 1 to load modules in their own
                       global contexts.
NODE_DISABLE_COLORS  Set to 1 to disable colors in the REPL
Documentation can be found at http://nodejs.org/ or with 'man node'
```

这就输出了与使用方法相关的信息，展示可用的命令行选项。

注意，同时支持Node和V8的命令行选项（没有在上面的命令行中显示）是存在的。记住，Node是以V8引擎为基础的，而V8含有自己的选项域，主要针对字节码编译或者垃圾回收的细节和堆算法。输入`node --v8 -options`可以看到完整的命令列表。

在使用命令行工具时，你可以指定命令行的选项、单个脚本文件和传入脚本的参数列表。我们会在下一节提到与脚本参数相关的内容。

不带参数运行Node时，就会进入JavaScript shell会话程序：

```
$ node
> console.log('Hello, world!');
Hello, world!
> console.log(JSON.stringify(require.paths));
["/Users/davidherron/.node_libraries","/opt/local/lib/node"]
```

所有可写入Node脚本的代码都可以直接写在这里。命令解释器提供了良好的终端体验，这对你玩代码非常有帮助。你也玩代码，对吧？很好。

2.4.2 用 Node 运行简单的脚本

现在让我们看看如何用Node运行脚本。这个操作非常简单，让我们从帮助命令开始：

```
$ node --help
Usage: node [options] script.js [arguments]
```

上面展示了一个脚本文件名和一些传入脚本的参数，这与其他脚本语言很类似。

首先创建一个文本文件ls.js，输入下面的内容：

```
var fs = require('fs');
var files = fs.readdirSync('.');
for (fn in files) {
  console.log(files[fn]);
}
```



下载示例代码

你可以下载所有已购Packt图书的示例代码文件，只要你是在http://www.PacktPub.com上用你的账号购买的。如果你在其他地方购买了本书，可以访问http://www.PacktPub. com/support并注册，代码文件会以邮件的形式发送给你。

接下来输入下面的命令：

```
$ node ls.js
app.js
ls.js
```

这个命令比较简单地仿制了Unix系统里ls命令的功能。readdirSync函数和Unix里的readdir命令功能相似（输入man 3 readdir了解更多），这个函数可以列出目录下的所有文件。

脚本的参数会通过一个全局的数组process.argv传递，你可以按照下面的方式修改ls.js，查看这个数组的工作原理：

```
var fs = require('fs');
var dir = '.';
if (process.argv[2]) dir = process.argv[2];
var files = fs.readdirSync(dir);
for (fn in files) {
  console.log(files[fn]);
}
```

你可以按照下面的方式运行这段脚本：

```
$ node ls2.js ../0.4.8/bin
node
node-waf
```

2.4.3　用 Node 启动服务器

以后你运行的脚本大部分都可能是服务器进程。接下来我们也会运行很多这类脚本。因为我们现在只是在验证和熟悉Node，所以这里只给出一个简单的HTTP服务器。这里借用了Node主页（http://nodejs.org）上的服务器脚本。

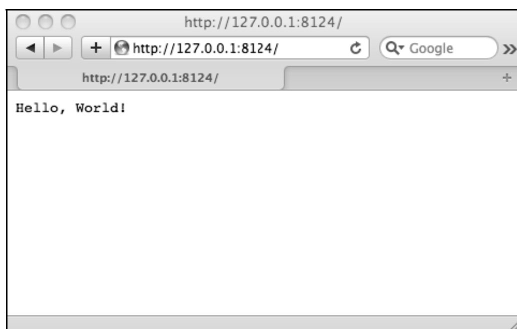
创建一个文件app.js，包括了下面的内容：

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
}).listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124');
```

按照下面的方式运行：

```
$ node app.js
Server running at http://127.0.0.1:8124
```

这是一个用Node构建的、最简单的Web服务器。如果你对它的工作原理比较感兴趣，可以直接阅读第4章～第6章。现在，你只需要在浏览器里访问http://127.0.0.1:8124就可以看到如图所示的结果：



这里有一个问题，为什么这段脚本执行完毕后`ls.js`退出了，而这段脚本却没有退出？两个脚本都执行到了最后部分；在`app.js`里Node进程没有退出，而`ls.js`进程退出了。原因在于活动的事件监听器。Node始终会启动一个事件循环，在`app.js`里，`.listen`函数创建了一个实现HTTP协议的事件监听器。这个事件监听器会使app.js一直处于执行状态，直到你在终端窗口中执行一些退出操作，比如Ctrl+C。但是在`ls.js`脚本里并不能创建一个长时间运行的事件监听器，所以当执行到脚本末尾的时候，Node进程就会退出。

2.5　安装 npm——Node 包管理器

Node本身是一个非常基础的系统，只是包含了一个JavaScript解释器和一些有趣的异步I/O库。第三方Node模块的生态系统迅速发展是Node引人注意的原因之一，而整个生态系统的中心就是

npm。这些模块可以以源代码的形式下载并手工添加用到Node程序中，而npm提供了一个更简单的方式。npm是一个非官方标准的Node包管理器，它极大地简化了下载和使用Node模块的流程。我们会在下一章详细介绍npm。

输入npmjs.org主页上展示的如下命令来安装npm：

```
$ curl http://npmjs.org/install.sh | sh
```

这个命令会在你的系统中下载并执行一个shell脚本，或许你需要在执行之前先考虑用下面的命令检查系统和这一shell脚本的兼容性。

```
$ curl http://npmjs.org/install.sh | less
```

这个命令会安装npm脚本和包到Node所安装的源文件中。这意味着你需要注意两个地方来保证安装的正确性。

如果需要通过修改`PATH`环境变量运行Node，你可以通过下面的方式在运行npm安装程序时检查`PATH`环境变量的设置是否正确：

```
$ export PATH=/path/to/node/0.n.y/bin:${PATH}
$ curl http://npmjs.org/install.sh | sh
```

下一个要考虑的是假设需要通过`sudo make install`命令将Node安装在系统级目录。那么，npm的安装需要这样完成：

```
$ curl http://npmjs.org/install.sh | sudo sh
```

使用`sudo sh`意味着安装npm的进程（/bin/sh）是运行在root权限下的。

现在npm已经安装完成，让我们尝试使用一下：

```
$ npm install -g hexy
/home/david/node/0.4.7/bin/hexy ->
/home/david/node/0.4.7/lib/node_modules/hexy/bin/hexy_cmd.js
hexy@0.2.1 /home/david/node/0.4.7/lib/node_modules/hexy
$ hexy --width 12 ls.js
00000000: 7661 7220 6673 203d 2072 6571  var.fs.=.req
0000000c: 7569 7265 2827 6673 2729 3b0a  uire('fs');.
00000018: 7661 7220 6669 6c65 7320 3d20  var.files.=.
00000024: 6673 2e72 6561 6464 6972 5379  fs.readdirSy
00000030: 6e63 2827 2e27 293b 0a66 6f72  nc('.');.for
0000003c: 2028 666e 2069 6e20 6669 6c65  .(fn.in.file
00000048: 7329 207b 0a20 2063 6f6e 736f  s).{...conso
00000054: 6c65 2e6c 6f67 2866 696c 6573  le.log(files
00000060: 5b66 6e5d 293b 0a7d 0a         [fn]);.}.
```

再说明一下，我们会在下一章深入探讨npm。`hexy`既是一个Node库，也是一个用于输出十六进制数据的脚本。

2.6 系统启动时自动启动 Node 服务器

之前我们用命令行工具启动了Node服务器。虽然这对测试和开发有帮助，但是对平时部署应

用毫无用处。启动服务进程的方式有很多，具体因操作系统不同而异。实现一个Node服务器意味着这个服务器会和其他后台进程（sshd、apache、mysql等）一样使用，例如启动/停止脚本。

Node项目没有为任何操作系统提供启动/停止脚本。也许有人会说，如果Node包含了这两个脚本，它就做了一个平台不该涉及的事。但是对于Node服务器应用来说，它们需要有这样的脚本。传统的方式是使用init守护进程管理后台进程，这是通过调用/etc/init.d目录下的脚本实现的。Fedora和Redhat还是使用了这个进程，但其他操作系统使用了其他守护进程管理器，比如Upstart或者launchd。

编写的这些启动/停止脚本只是服务器必需的一部分。Web服务器需要做到可靠（例如在崩溃时能够做到自动重启）、易于管理（和系统管理实践结合得很好）、直观（将STDOUT输出内容保存到记录文件中）等。Node更像是一个建筑工具箱，包含了构建服务器的各个“零件”，本身并不是一个完整而优美的服务器。基于Node实现一个完整的Web服务器意味着你需要综合操作系统本身的后台进程管理服务，实现需要的记录功能，实施安全措施或者防御措施来对抗不良行为，比如阻断来自外网的服务攻击等。

这里有一些工具或者措施，用于结合Node服务器与各个操作系统的后台进程管理器，保证在系统启动后服务器能可持续运行。我们马上就会实现一些简单的策略让服务器能长久运行在Debian服务器上。下面是一些在不同的平台上将Node作为后台守护进程执行的方式。

- nodejs-autorestart（https://github.com/shimondoodkin/nodejs-autorestart）通过Upstart（Ubuntu、Debian等系统上）管理Linux上的Node实例。
- fugue（https://github.com/pgte/fugue）会监听一个Node服务器，在其崩溃的时候重启它。
- forever（https://github.com/indexzero/forever）是一个小型的命令行Node脚本，用于确保一个脚本能持续运行。Charlie Robbins写了一篇博文讲解了它的使用（http://blog.nodejitsu.com/ keep-a-nodejs-server-up-with-forever）。
- Node-init（https://github.com/frodwith/node-init）是一个Node脚本，它可以将Node应用转换成符合LSB的启动脚本。LSB是Linux兼容性的规范之一。
- Debian系统上的launchtool（http://people.debian.org/~enrico/launchtool. html）是一个系统级命令，用于监听任一命令的启动，包括将其作为一个守护进程运行。
- Ubuntu上的Upstart工具（http://upstart.ubuntu.com/）可以单独使用（http://caolanmcmahon.com/posts/deploying_node_js_with_upstart）或者和monit（http://howtonode.org/deploying-node-upstart-monit）一起使用来管理Node服务器。
- 在Mac OS X上，有人写了一个`launchd`脚本。苹果已经在http://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Articles/LaunchOnDemandDaemons.html 上发布了一个实现`launchd`脚本的说明。

实践是检验真理的唯一标准，接下来我们尝试使用forever工具，并结合一个LSB标准的启动脚本实现一个小型的Node服务器进程。服务器搭建在基于VPS技术的Debian系统上，Node和npm安装在/usr/local/node/0.4.8目录上。下面的服务器脚本位于/usr/local/app.js（这个不是最佳的位置，但是对这个示例有些用处）。

```
#!/usr/bin/env node
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337);
```

仔细注意脚本的第一行。这是Unix/POSIX系统中的一个小窍门，可以让脚本变得可执行。

forever工具是按照下述方式安装的：

```
$ sudo npm install -g forever
```


forever可以管理后台进程。它可以在进程崩溃的时候重启进程，传递标准输出和错误流到记录文件，并且有其他一些实用的特性。这个工具值得探究。

最后一步是设置/etc/init.d/node脚本，它是从另一个/etc/init.d 脚本修改而来：

```
#! /bin/sh -e
set -e
PATH=/usr/local/node/0.4.8/bin:/bin:/usr/bin:/sbin:/usr/sbin
DAEMON=/usr/local/app.js
case "$1" in
  start) forever start $DAEMON ;;
  stop)  forever stop  $DAEMON ;;
  force-reload|restart)
    forever restart $DAEMON ;;
  *) echo "Usage: /etc/init.d/node {start|stop|restart|force-reload}"
     exit 1
     ;;
esac
exit 0
```

在Debian系统中，你可以用下面的命令设置一个`init`脚本：

```
$ sudo /usr/sbin/update-rc.d node defaults
```

这句命令会配置你的系统，从而让/etc/init.d/node脚本可以在系统重启和关闭时启动和停止后台进程。每一次系统启动或者关闭的时候，每个`init`脚本都会执行，第一个参数是`start`或者`stop`。因此，当`init`脚本在系统启动或关闭过程中执行时，下面两行代码之一就会执行。

```
start) forever start $DAEMON ;;
stop)  forever stop  $DAEMON ;;
```

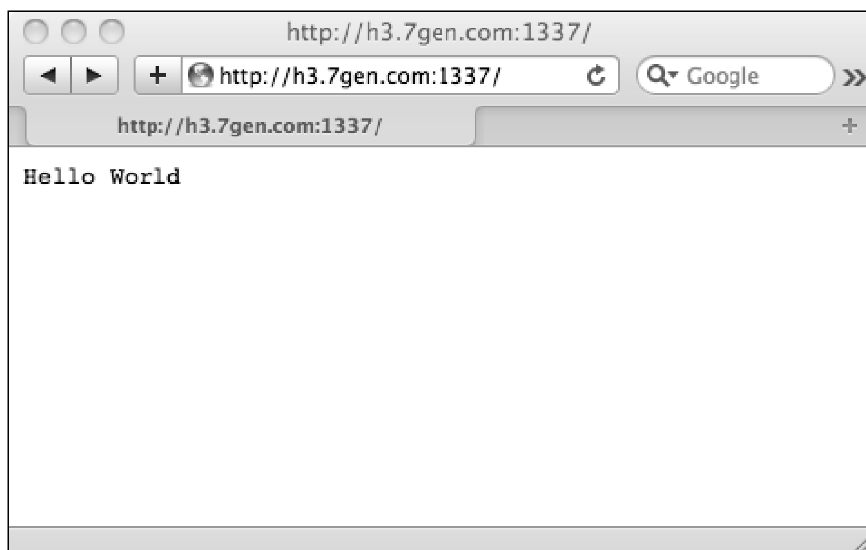
我们可以手动执行`init`脚本：

```
$ sudo /etc/init.d/node start
info: Running action: start
info: Forever processing file: /usr/local/app.js
```

因为`init`脚本使用了forever工具，我们可以从forever中获取所有由它启动的进程的状态。

```
$ sudo forever list
info: Running action: list
info: Forever processes running
  [0] node /usr/local/app.js [16666, 16664] /home/me/.forever/7rd6.log 0:0:1:24.817
```

当服务器进程在服务器上运行时，你可以打开浏览器进行访问，如下图所示。



因为服务器处于执行状态并由forever管理，所以我们可以直接查看下面这些进程：

```
$ ps ax | grep node
16664 ? Ssl 0:00 node /usr/local/node/0.4.8/bin/forever start /usr/local/app.js
16666 ? S   0:00 node /usr/local/app.js
```

最后，你可以按照下面的方式关闭脚本：

```
$ sudo /etc/init.d/node stop
info: Running action: stop
info: Forever processing file: /usr/local/app.js
info: Forever stopped process:
  [0] node /usr/local/app.js [5712, 5711] /home/me/.forever/Gtex.log 0:0:0:28.777
$ sudo forever list
info: Running action: list
info: No forever processes running
```

如何在多核系统中调用所有 CPU

V8是一个单线程的JavaScript引擎。对于Chrome浏览器来说，这样已经足够了，但是如果服务器是基于Node开发的，我们就只能使用一个新的16核服务器中的一个CPU，其他15个CPU却闲置在一边。你的上司或许会希望你能将其他15个CPU也利用起来。

一个单线程进程只能使用一个CPU。这就是事实。在一个线程中使用多核需要支持多线程的软件。但是Node是“没有线程概念”的平台，虽然这让编程变得简单，但是也意味着Node不能将多核系统利用起来。该怎么做呢？或者更重要的是，如何取悦你的上司？

现在有很多项目在努力定制足够可靠的多线程Node,同时希望将多核系统的硬件性能都发挥出来。

最基础的想法就是启动多个Node进程，在进程之间共享请求通信量。使用多个单线程进程可以充分利用所有的CPU，并让你的上司觉得服务器的投入是值得的。

Cluster（https://github.com/ LearnBoost/cluster）是多线程Node服务器项目中的一个，它被称为“为Node.js设计的、可扩展的多核服务器管理器”。它会启动一系列可配置的子进程，在进程崩溃的时候重启它们，并拥有可扩展的记录功能，命令控制模块和统计功能。老的Spark项目已停止，人们开始转投Cluster项目。

Cluster项目包括了几个服务器配置示例，展示了它的作用。让我们安装Cluster并使用一个例子看看它的工作原理：

```
$ sudo npm install cluster
cluster@0.6.4 ./node_modules/cluster
└─── log@1.2.0
```

我们要使用的例子（reload.js）是其中的一个典型，我们对app.js进行修改并创建cluster-app.js，其中包含下面的代码：

```
#!/usr/bin/env node
var http = require('http');
var cluster = require('cluster');
var server = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
})
cluster(server).set('workers', 2).use(cluster.reload())
  .listen(1337);
```

Cluster配置创建了两个工作者进程并用于共享负荷，它会自动重新加载已修改的文件。你可以阅读Cluster项目站点上的文档了解更多内容。

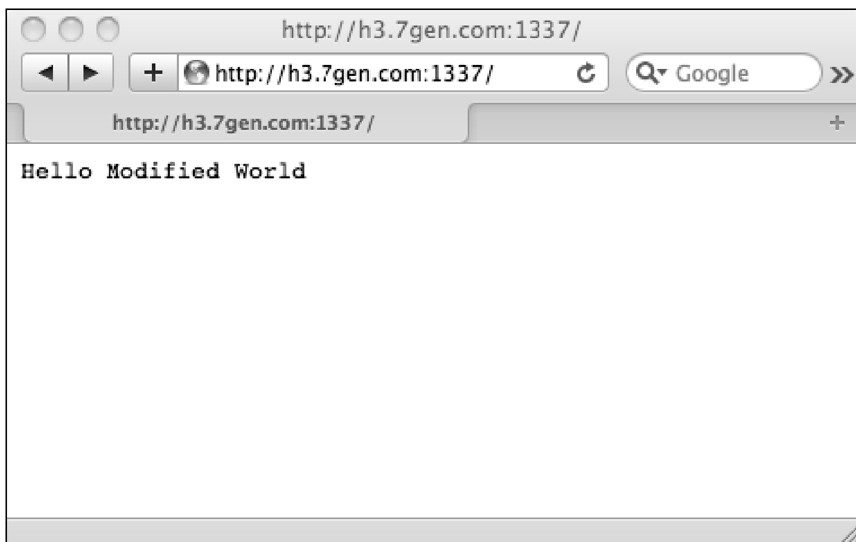
它可以通过node cluster-app.js命令运行，不过我们要通过修改/etc/init.d/node来运行这个脚本。我们只需要设置DAEMON变量为如下的值：

```
DAEMON=/usr/local/cluster-app.js
```

然后：

```
$ sudo /etc/init.d/node start
info: Running action: start
info: Forever processing file: /usr/local/cluster-app.js
$ sudo forever list
info: Running action: list
info: Forever processes running
  [0] node /usr/local/cluster-app.js [6522, 6521]
$ ps ax | grep node
6521 ? Ssl 0:00 node /usr/local/node/0.4.8/bin/forever start
/usr/local/cluster-app.js
 6522 ? Sl  0:15 node /usr/local/cluster-app.js
 6541 ? S   0:00 /usr/local/node/0.4.8/bin/node  /usr/local/cluster-app.js
 6542 ? S   0:00 /usr/local/node/0.4.8/bin/node /usr/local/cluster-app.js
```

现在你可以看到一个多进程的Node服务器正在运行。我们可以用ps命令查看两个进程，也可以通过浏览器访问http://example.com:1337/来进行验证服务器是否正在运行（服务器运行时会看到“Hello World”消息）。但是因为它使用了Cluster的自动重新加载特性，你可以对Cluster-app.js做一些适当的修改，使得刷新浏览器（不需要重启服务器）时可以看到如下图所示的内容。



2.7 小结

我们在这一章学习了Node的安装、命令行工具的使用和如何运行一个Node服务器。我们也介绍了一些之后几章会详细介绍的内容。

本章具体内容包括：

- 下载并编译Node源代码；
- 安装Node的方法，分别针对个人开发的用户目录安装方式和针对部署的系统级安装方式；
- 安装非官方的Node包管理器npm；
- 运行Node脚本和Node服务器；
- 如何更安全地用Node启动后台进程；
- 如何用Node实现多进程，以利用所有CPU。

现在我们已经了解如何配置基本的系统，接下来可以用Node开发应用了。首先我们会学习以模块为单位的Node应用构建方式，这是下一章的主要内容。

第 3 章

Node模块

3

开始编写Node应用之前，必须先学习Node的模块和包。模块和包是组成应用的基本单位。

本章内容

- 什么是模块；
- CommonJS模块规范；
- Node搜索模块的方式；
- npm包管理系统。

开始吧。

3.1　什么是模块

模块是构成Node应用的组件。实际上我们已经看到过一些模块了——我们在Node中使用的每一个JavaScript文件都是一个模块。接下来让我们看看这些模块是什么，它们是怎么工作的。

在第2章的`ls.js`例子里，我们写了如下的代码来引入`fs`模块，从而可以访问`fs`模块内部的函数：

```
var fs = require('fs');
```

`require`函数会搜索模块，然后将模块的定义载入Node的执行环境，从而让对应的函数可以供外部调用。这个例子里的`fs`对象包含从`fs`模块里导出的代码和数据。

在进行详细介绍前，让我们先看一个简单的例子。思考一下模块`simple.js`：

```
var count = 0;
exports.next = function() { return count++; }
```

这个模块定义了一个对外开放的函数和一个局部变量。使用方法如下图所示。

从`require('./simple')`返回的这个对象正是我们在`simple.js` 里定义的`exports`。每一次对`s.next` 的调用都是对`simple.js`里`next`函数的调用，然后`next`函数返回（已自增的）`count`变量值，这样也就解释了为什么`s.next`方法返回的值越来越大。

`require`的机制就是`exports`的所有成员（包括函数、对象）都会被暴露给模块外部的代码，而那些没有指定在`exports`下的对象，在模块外部是不可见的。这就是一个模块封装的例子。

```
Terminal — bash — 71×17
$ node
> var s = require('./simple');
> s.next();
0
> s.next();
1
> s.next();
2
> s.next();
3
> $ 
```

我们已经简单了解模块的封装，接下来进行更深入的了解。

3.1.1　Node 模块

Node的模块实现深受CommonJS模块规范（会在本章最后做介绍）的影响，但并不与之相同。只有当你想要在Node和其他CommonJS系统之间共享代码时，它们之间的区别才会显得比较重要。只要快速浏览一下Modules/1.1.1规范就能知道，两者之间的差别并不重要，而且就我们的目的而言，只要学习怎样使用Node就够了，不用在两者的差异上想太多。

3.1.2　Node 解析 `require('module')`的方式

在Node里，模块存储在文件中，每个文件仅存储一个模块。在文件系统中有几种方法命名和部署模块，这些方法灵活多样，尤其是使用非官方的Node包管理器npm[1]时。

1. 模块标识符与路径名

一般来说，模块名就是一个没有文件后缀的路径名。也就是说，当我们写下`require('./simple')`时，Node会自动把`.js`加到文件名后并加载`simple.js`。

那些以“`.js`”为后缀名的文件对应的模块自然应该以JavaScript编写。Node也支持二进制原生语言库，并将它们用作Node模块。这样的情况下，文件名后缀就是.node。不过，我们不会探讨如何实现原生语言的Node模块，但当你遇到这些模块时这些内容足以让你理解和辨别它们。

一些Node模块并不以文件形式存储在操作系统的文件系统中，而是会被编译成Node可执行文件，这些模块是核心模块，nodejs.org官方文档已经列出。它们原本在Node源代码中以文件形式存在，但是构建进程将其编译成了二进制Node可执行文件。

Node里有3种定义模块的方式：相对路径的定义方式、绝对路径的定义方式和顶级目录的定义方式。

用相对路径定义的模块的标识符以“`./`”或者“`../`”开头，用绝对路径定义的模块的标识

① 现在npm已经被集成到node安装包里了。

符以“/”开头。这与POSIX文件系统中通过文件路径执行可执行文件的语法是一致的。

很明显，绝对路径定义的模块标识符与文件系统的根目录相关。

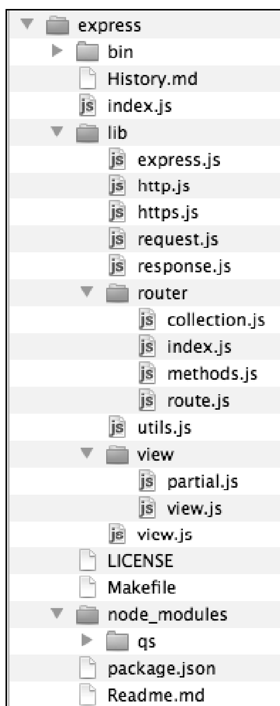
顶级目录的模块标识符不是以“.”、“..”或“/”开头，而是以模块名开头。这些模块储存于某个目录中，比如node_modules目录，或者其他被列在require.paths[①]数组（Node将之用于指定存储这些模块的目录）里的目录中。这些内容会在之后讨论到。

2. Node应用里的本地模块

所有模块被整齐地划分为两类，一类是特定应用的模块，另一类就是非特定应用的模块。但愿非特定应用的模块会被写成通用模块。我们现在开始介绍如何实现应用要使用的模块。

应用通常都会有一个专属的目录结构，遵从这一目录结构的模块文件按序放置在源代码控制系统中，然后会被部署到服务器上。这些模块能够获取该应用内其他模块的相对路径，并能通过相对路径标识符互相访问。

举个例子，为方便理解，让我们看下一个已经构建好的Express应用框架的Node包目录。它包括了以多级目录形式存放的模块，Express开发者比较喜欢这样的组织方式。你可以想象下为应用创建一个达到一定复杂程度的层级，将应用分解成一个个比模块大但比应用小的数据块的情形。遗憾的是在Node里没有词可以形容它，所以我们只能用诸如“分解为比模块大的数据块”之类的笨拙语句来形容。每个被分解出来的数据块都可以作为含有多个模块的目录来实现。



① 现在require.paths已经被移除了。

在这个例子里，我们最有可能引用模块utils.js。我们可以使用如下的require语句之一使用utils.js：

```
var utils = require('./lib/utils');
var utils = require('./utils');
var utils = require('../utils');
```

3. 绑定应用的外部依赖

引用node_modules目录中的模块必须使用顶级目录标识符：

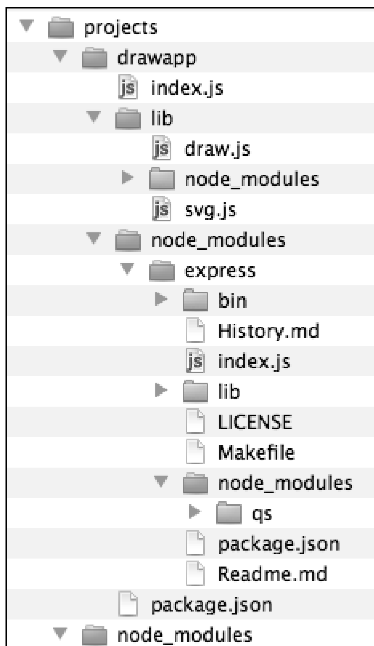
```
var express = require('express');
```

Node会在寻找模块时搜索node_modules目录。Node里含有不止一个以供搜索的node_modules目录。Node会从当前模块的目录开始，先把node_modules目录添加进去，然后搜索当前目录下的node_modules。如果在当前node_modules目录下没有找到对应的模块，Node会转向父级目录并再次进行搜索，以此类推，直到到达文件系统的根目录。

在前面的例子里，注意node_modules文件夹中有一个名为qs的目录。在这个目录位置，若执行下面的代码，Express里的所有模块都可以访问qs模块。

```
var qs = require('qs');
```

如果你想要在应用中使用Express框架，该怎么办？很简单，在你的应用目录里创建一个node_modules目录，然后把Express框架安装到那里，如下图所示。



如上图所示，假设有一个应用drawapp。因为node_modules处于drawapp中的一个子目录下，所以所有在drawapp里的模块都可以用下面的代码访问express。

```
var express = require('express');
```

但是这些模块无法访问存放在express模块内的qs模块。在文件系统中，对node_modules目录的搜索是往父级（逐级）查找，而不是查找子目录。

同样，一个安装在lib/node_modules目录下的模块也可以被draw.js或者svg.js访问，但是不能被index.js访问。对node_modules目录的搜索都是逐层往上而不是逐层往下进行的。

Node会逐层向上搜索node_modules目录，然后在第一次找到需要的模块后停止在所在目录。draw.js或者svg.js内部的一个模块引用会搜索如下的目录：

- /home/david/projects/drawapp/lib/node_modules
- /home/david/projects/drawapp/node_modules
- /home/david/projects/node_modules
- /home/david/node_modules
- /home/node_modules
- /node_modules

在维护多个版本冲突的Node包时，node_modules目录起着关键作用。比起只用一个地方存放模块，然后因多个版本的模块冲突抓狂，更好的方式就是提供多个node_modules目录，以便必要时在特定的位置存放特定版本的模块。若将不同版本的同一模块放在不同的node_modules目录下，只要node_modules目录位置正确，模块间就不会相互冲突。

比如，如果写了一个使用forms模块（https:// github.com/caolan/forms）辅助创建表单的应用，在你写了几百个不同的表单之后，forms模块的作者对模块进行了一些修改，而这些修改可能使模块与你的应用不兼容。面对这么多需要转换和测试的表单，你可能不会想一下子完成这一工作。那么，这样的情况下，你就需要为应用建两个目录，每一个目录都需要有属于自己的node_modules文件夹，文件夹里对应存放不同版本的forms模块。当成功将一个表单转换到新forms模块时，你就可以将那些代码转移到新forms模块的文件夹下。

4. require.paths目录下的系统级模块

Node用来查找node_modules目录的算法会搜索应用源代码树之外的模块。它会一直搜索，直到文件系统的根目录下，所以你最好能有一个包含全局模块的node_modules目录，以满足Node对模块的任意搜索。

Node还通过require.paths变量提供了一项功能。它是一个由目录名构成的数组，我们可以在这个数组里搜索模块。

具体的例子如下：

```
$ node
> require.paths;
["/home/david/.node_modules","/home/david/.node_libraries","/usr/local/lib/node"]
```

NODE_PATH环境变量也可以添加目录到require.paths数组中：

```
$ export NODE_PATH=/usr/lib/node
$ node
> require.paths;
["/usr/lib/node","/home/david/.node_libraries","/usr/local/lib/node"]
>
```

曾几何时，人们习惯于按照下面这种方法给require.paths添加目录：require.paths.push(_dirname)。不过，现在这已经不是推荐的方式，因为实践证明这个方法会导致一些麻烦。虽然，现在你依然可以使用这样的方法，而且依然存在使用这个习惯构建的模块，但我们绝不赞成这么做。当将多个模块的目录放进require.paths时，结果会变得不可预知。

大部分情况下，最佳实践就是将模块放到node_modules目录里。

5. 复杂的模块——作为目录的模块

一个复杂的模块可能包括了许多内部模块、数据文件、模版文件、文档、测试文件等。这些内容可以存放在一个仔细构建的目录结构里，由Node将其视作一个模块，可以满足一个require('moduleName')这样的请求的处理需要。为此，你先得把index.js模块文件或者package.json文件放到一个目录里。package.json文件里包含描述模块的数据，其数据格式和npm定义的package.json格式基本一致（稍后介绍）。这两者都使用了npm能识别的非常小的标签子集，从而做到了与Node兼容。

具体而言，Node可以识别package.json里这样类型的字段：

```
{ name: "myAwesomeLibrary",
  main: "./lib/awesome.js" }
```

有了package.json之后，require('myAwesomeLibrary')代码就可以找到这个目录并加载对应的文件：

```
/path/to/node_modules/myAwesomeLibrary/lib/awesome.js
```

如果没有package.json文件，Node会查找index.js，从而加载文件：

```
/path/to/node_modules/myAwesomeLibrary/index.js
```

无论选择了index.js还是package.json，含有内部模块和其他资源的复杂模块都是比较容易实现的。回头来看我们之前讨论的Express包结构，一些模块会使用模块的相对路径标识符引用包里的其他模块，而你可以使用一个node_module目录整合在其他地方开发的模块。

3.2　Node 包管理器

就像在第2章里描述的，npm是一个Node包管理和分发工具。它已经成为了非官方的发布Node模块（包）的标准。从概念上理解，它和apt-get（Debian）、rpm/yum（Redhat/Fedora）、MacPorts（Mac OS X）、CPAN（Perl）或者PEAR（PHP）这些工具很类似。它存在的目的就是在因特网上通过简单的命令行界面发布和管理Node包。有了npm，你就可以很快地找到特定服务要使用的包，进行下载、安装以及管理已经安装的包。

npm为Node定义的包格式大部分基于CommonJS包规范。

3.2.1　npm 包的格式

一个npm包是包含了package.json的文件夹，package.json描述了这个文件夹的结构。除了npm所识别的package.json标签比Node能识别的多得多，它恰好就是我刚刚说到的复杂模块。CommonJS Packages/1.0 规范是npm的package.json的起点。npm的package.json文档可以通过如下方式访问：

```
$  npm help json
```

一个基本的package.json文件是这样的：

```
{ name: "packageName",
  version: "1.0",
  main: "mainModuleName",
  modules: {
    "mod1": "lib/mod1",
    "mod2": "lib/mod2"
  }
}
```

这个文件是JSON格式的，作为一名JavaScript程序员，这样的格式你应该已经很熟悉了。

最重要的标签就是name和version。名称会出现在URL还有命令名里，所以需要选择一个对双方都可靠的。如果想要在公共的npm仓库里发布一个包，你最好在http://search.npmjs.org上（或通过如下命令）检查包名是否已被使用。

```
$ npm search packageName
```

main标签的使用和在前一节关于复杂模块的讨论类似。当调用require('packageName')时，main指向的入口模块会返回。包本身也可以包含很多模块，这些模块都可以在modules列表中列出。

Node包是可以以tar-gzip格式打包的，特别是要将Node包通过因特网传输时。

Node包可以声明与其他包的依赖关系。这样的话，npm可以自动安装其他被依赖的包。依赖关系是这样定义的：

```
"dependencies":
  { "foo" : "1.0.0 - 2.9999.9999"
    , "bar" : ">=1.0.2 <2.1.2"
  }
```

描述和关键词字段可以帮助人们在npm仓库（http://search.npmjs.org）中寻找Node包。Node包的所有权信息可以记录在主页、作者或者贡献者字段中：

```
"description": "My wonderful packages walks dogs",
  "homepage": "http://npm.dogs.org/dogwalker/",
  "author": dogwhisperer@dogs.org
```

一些npm包通过向用户的PATH变量写入数据（路径）提供可执行程序。这些包是通过bin标签声明的。这是一个命令名到实现该命令的脚本的映射。命令脚本会被通过名称引用并安装在包含Node可执行文件的目录中。

3

```
bin: {
  'nodeload.js': './nodeload.js',
    'nl.js': './nl.js'
},
```

directories标签用于记录包的目录结构。lib目录中的文件会被自动扫描并加载。现在还有其他特定于二进制文件、指南和参考文件这样的目录字段。

```
directories: { lib: './lib', bin: './bin' },
```

脚本标签是一个在Node包生命周期中对应各个事件执行的脚本命令。这些事件包括install、activate、uninstall、update等。你可以用下面的命令查看更多的脚本命令。

```
$ npm help scripts
```

以上内容只是关于npm包格式的一小部分，欲知更多详细资料可以阅读文档（npm help json）。

3.2.2　查找 npm 包

默认情况下，npm模块可从http://npmjs.org官方网站上的安装包登记处获取。[①]如果你知道这个模块的名字，可以通过输入下述命令来安装：

```
$ npm install moduleName
```

但是如果你不知道模块的名字怎么办？如何才能找到想要的模块？

http://npmjs.org网站会把已注册的模块按照索引公布出来，这样你就可以在http://search.npmjs.org网站上按照索引值找到想要的模块。

npm也提供了查找模块的命令行搜索功能：

```
$ npm search mp3
mediatags  Tools extracting for media meta-data tags  =coolaj86  util m4a aac mp3 id3
jpeg exiv xmp
node3p     An Amazon MP3 downloader for NodeJS.       =ncb000gt
```

当然，上面找到的模块可以按照如下方式安装：

```
$ npm install mediatags
```

安装一个模块后，有些人可能会想看一下模块对应站点上的文档。package.json文件里的homepage标签对应的值就是这个站点主页的地址。查看package.json文件最方便方法就是如下使用命令npm view：

```
$ npm view zombie
...
{ name: 'zombie',
  description: 'Insanely fast, full-stack, headless browser testing using Node.js',
...
  version: '0.9.4',
```

① 这里的言下之意应该是说可以自己架设包服务器。

```
  homepage: 'http://zombie.labnotes.org/',
...
npm ok
```

你可以使用npm view从package.json里查看任意一个标签对应的信息，比如下面的命令能用于查看homepage标签：

```
$ npm view zombie homepage
http://zombie.labnotes.org/
```

3.2.3 使用 npm 命令

npm主命令包含很多子命令，子命令对应特定的包管理操作。这些命令涉及一个Node包生命周期的每一个部分操作，包括发布（站在包作者的角度）、下载、使用或者移除（站在一个npm使用者的角度）。



1. 获取npm的帮助

或许最重要的事情是知道去哪里寻求帮助。最主要的帮助信息可以通过下述方式获取。

你可以按照下面的方式查看大部分命令的帮助信息：

```
$ npm help <command>
```

npm网站（http://npmjs.org/）有FAQ（常见问题解答），并且提供npm软件下载。或许和npm相关的最重要问题（及答案）就是：**为什么npm不喜欢我**？npm不懂恨，它对每一个人都很友好，包括你。

```
Terminal — bash — 71×15
$ npm help

Usage: npm <command>

where <command> is one of:
    adduser, author, bin, c, cache, completion, config,
    deprecate, docs, edit, explore, faq, find, get, help, i,
    info, init, install, la, link, list, ll, ln, ls, outdated,
    owner, prefix, prune, publish, r, rb, rebuild, restart, rm,
    root, run-script, s, se, search, set, start, stop, tag,
    test, un, uninstall, unlink, unpublish, up, update, version,
    view, whoami

Add -h to any command for quick help.
```

2. 查看包信息

`npm view`命令会把`package.json`文件当做数据，让你能够找到使用点标记法记录的JSON标签，比如如下查看包的依赖关系：

```
$ npm view google-openid dependencies
{ express: '>= 0.0.1',
  openid: '>= 0.1.1 <= 0.1.1' }
```

`package.json`文件可以包含Node包仓库的地址（URL）。因此，如果想要检索包的源文件，可以使用如下代码：

```
$ npm view openid repository.url
git://github.com/havard/node-openid.git
$ git clone git://github.com/havard/node-openid.git
Cloning into node-openid...
remote: Counting objects: 253, done.
remote: Compressing objects: 100% (253/253), done.
remote: Total 253 (delta 148), reused 0 (delta 0)
Receiving objects: 100% (253/253), 63.29 KiB, done.
Resolving deltas: 100% (148/148), done.
```

对于一个包，什么版本的Node是必需的？

```
$ npm view openid engines
node >= 0.4.1
```

3. npm包的安装

`npm install`命令让安装Node包变得很容易：

```
$ npm install openid
openid@0.1.6 ./node_modules/openid
$ ls node_modules/
openid
```

仔细看你会发现Node包是安装在本地的`node_modules`目录下的。你也可以通过改变当前的目录将Node包安装到其他位置，或者让npm执行一个全局安装。例如，下面的命令会建立一个目录/var/www，其中/var/www/node_modules会存放为多个站点共享的模块。

```
$ cd /var/www
$ npm install openid
openid@0.1.6 ./node_modules/openid
```

npm的安装分为全局模式和本地模式。一般情况下会以本地模式运行，包会被安装到和你的应用代码同级的本地node_modules目录下。在全局模式下，Node包会被安装到Node的安装目录下（require.paths中的目录），而不是当前文件夹的子文件夹node_modules中。

全局模式下安装Node包的第一种方法是如下使用-g标志：

```
$ npm install -g openid
openid@0.1.6 /usr/local/node/0.4.7/lib/node_modules/openid
$ which node
/usr/local/node/0.4.7/bin/node
```

全局模式下的Node包安装位置取决于Node被安装的位置。

全局模式下的第二个安装方法是修改npm配置设置。配置设置里有很多配置项，这些配置项会在之后讨论，现在只探讨下面这个配置：

```
$ npm set global=true
$ npm get global
true
$ npm install openid
openid@0.1.6 /usr/local/node/0.4.7/lib/node_modules/openid
```

欲了解npm使用的所有文件夹，请键入下面的命令：

```
$ npm help folders
```

4. 使用已安装的Node包

安装Node包的意义在于启用Node程序，通过下面的方式访问模块：

```
var openid = require('openid');
```

np所做的就是帮你顺利地完成这些工作。

有些Node包的内部模块对其他软件有帮助。比如，当前我们安装的openid模块包含一个base64 encode/decode模块，其他软件也可以引用这个模块：

```
var base64 = require('openid/lib/base64').base64;
```

但是，openid模块可能会对它的base64 encode/decode模块做一些改动，且这样会影响到你的应用。一些Node包对结构进行了调整，从而可以提供一组相关的、可用上述方式引用的子模块，并向外部的模块提供特定于这些子模块的、稳定的API。

5. 查看已安装的Node包

npm list命令可以列出已经安装的包，它会对当前的目录进行一次搜索。记住，Node的模块搜索是从代码执行的当前目录开始的。因此，搜索结果取决于你当前使用的目录，也就是取决于当前目录中node_modules中的内容。

如下图所示，注意在不同目录下列出的模块的变化。

```
Terminal — bash — 87×27

$ pwd
/Users/davidherron/Node.JS/chap03
$ npm list
/Users/davidherron/Node.JS
├── async@0.1.9
├─┬ connect@1.4.2
│ └── qs@0.1.0
├── ejs@0.4.2
├─┬ express@2.3.11
│ └── qs@0.1.0
└── mime@1.2.2
$ cd ../chap06
$ npm list
/Users/davidherron/Node.JS/chap06
├── colors@0.5.0
├── ejs@0.4.2
├─┬ express@2.3.6
│ ├── connect@1.4.1
│ ├── mime@1.2.2
│ └── qs@0.1.0
├── mongodb@0.9.4
├── mongoose@1.3.3
├── mongoq@0.0.4
├── openid@0.1.6
└── sqlite3@2.0.12
$ 
```

默认情况下，列表会以树状结构显示，上面这样的截图对于其他命令来说并不是特别有用。通过配置`parseable`可以让数据这样显示，以便于使用。

```
$ npm set parseable=true
$ npm list
/home/david/Node/chap06
/home/david/Node/chap06/node_modules/ejs
/home/david/Node/chap06/node_modules/express
/home/david/Node/chap06/node_modules/express/node_modules/connect
/home/david/Node/chap06/node_modules/express/node_modules/mime
/home/david/Node/chap06/node_modules/express/node_modules/qs
/home/david/Node/chap06/node_modules/mongodb
/home/david/Node/chap06/node_modules/mongoose
/home/david/Node/chap06/node_modules/sqlite3
```

6. Node包脚本

npm允许Node包脚本在包生命周期的不同时间自动执行。现在已有的生命周期事件是测试、启动、停止和重新启动。

一个npm包可以以下面的方式进行测试：

```
$ npm test <packageName>
```

启动、停止和重新启动这些生命周期事件没有一个固定的含义。明显的用途就是启动或者停止一个与Node包相关的后台进程。

7. 查看和编辑Node包的内容

npm含有一对查看/修改包内容的命令。比如，你可以在开发的时候使用这对命令读取包的源文件（理解Node包的作用），查看Node包的示例目录，或者修改一些测试补丁。

例子如下。

```
Terminal — bash — 93×22
$ pwd
/Users/davidherron/Node.JS/chap06
$ npm explore mongoose

Exploring /Users/davidherron/Node.JS/chap06/node_modules/mongoose
Type 'exit' or ^D when finished

bash-3.2$ pwd
/Users/davidherron/Node.JS/chap06/node_modules/nongoose
bash-3.2$ ls
History.md      README.md       examples        lib             support
Makefile        docs            index.js        package.json    test
bash-3.2$ cd examples/
bash-3.2$ ls
schema.js
bash-3.2$ exit
$ pwd
/Users/davidherron/Node.JS/chap06
$ 
```

就像我们在终端（命令输出）里看到的那样，`explore`命令产生了一个子shell程序，这个程序的当前目录是模块的安装位置。输入exit或者*control-D*会停止该shell程序，并返回到你之前登录时的shell程序。

如果有需要的话，你可以在浏览包内容的时候编辑文件。如果已经这么做了，你就需要像下面这样对包进行一次重建：

```
$ npm rebuild mongoose
mongoose@1.3.3 /home/david/Node/chap06/node_modules/mongoose
```

8. 更新已安装的Node包

开发者会经常更新他们的Node包，除非你跟上更新的进度，否则只能使用一些旧的特性。

下面的命令可用于检查包是不是已经过时：

```
$ npm outdated
express@2.3.6 ./node_modules/express current=2.3.3
mongoose@1.3.6 ./node_modules/mongoose current=1.3.3
```

它会展示当前已安装包的版本和线上npm仓库里最新的版本。更新已安装的Node包是非常容易的：

```
$ npm update express
connect@1.4.1 ./node_modules/express/node_modules/connect
mime@1.2.2 ./node_modules/express/node_modules/mime
qs@0.1.0 ./node_modules/express/node_modules/qs
express@2.3.6 ./node_modules/express
```

9. 卸载已安装的npm包

你最爱的Node包可能会变成你的噩梦，而且你还可能会有其他很多理由舍弃已经安装的Node包。你可以这样进行卸载：

```
$ npm list
/home/david/Node
└─── openid@0.1.6
$ npm uninstall openid
$ npm list
/home/david/Node
(empty)
```

10. 开发和发布npm包

现在我们已经了解了如何使用npm，接下来看一下npm包的开发。有一些npm命令可用于开发过程。

第一步是创建一个`package.json`文件，我们可以使用`npm init`命令创建初始版本的`package.json`。npm会问你几个问题，然后像下面这样迅速帮你创建如下内容：

```
{
  "author": "I.M. Awesome <awesome@example.com>",
  "name": "tmod",
  "description": "Test Module",
  "version": "0.0.1",
  "repository": {
    "url": ""
  },
  "engines": {
    "node": ">0.4.1"
  },
  "dependencies": {},
  "devDependencies": {}
}
```

第二步就是创建包源文件。npm没有提供帮你完成这个任务的方法。你是开发者，所以需要自己写代码。注意，增加一些内容到包里时，你需要更新`package.json`文件。不过npm还是提供了一些命令让你在开发Node包的时候使用。

其中一个命令是`npm link`，一个包的轻量级安装方法。和`npm install`方法的不同之处在于，`npm link`只是设置了一个连接到你的源文件目录的符号链接，然后你可以自由编辑包文件，而不需要在包有变化的时候重新打包和更新。你可以反复在一个包上修改和测试，不用经常重建。

`npm link`命令的使用分两个步骤，首先你要将自己的项目链接到Node安装程序上，如下所示：

```
$ cd tmod
$ npm link
../../0.4.7/lib/node_modules/tmod -> /home/david/Node/chap03/tmod
```

第二步，将Node包链接到你的应用里：

```
$ npm link tmod
../node_modules/tmod -> /home/david/Node/0.4.7/lib/node_modules/tmod ->
/home/david/Node/chap03/tmod
```

箭头（->）显示了由以上命令生成的符号链接链。

npm install安装命令在开发过程中有两种模式。第一种是在Node包目录的根目录下执行的时候，它会将当前目录和依赖关系安装到本地的node_modules目录。

第二种是利用一个本地文件或者通过一个URL利用网络安装压缩的Node包。大部分源代码控制系统支持一个以树形源代码打包的tarball文件（已经打包好的tar文件）。例如，在github项目上的下载页提供了一个这样的URL：

```
$ npm install https://github.com/havard/node-openid/tarball/v0.1.6
openid@0.1.6 ../node_modules/openid
```

当你对自己的Node包感到满意时，或许会希望将其发布到公共的npm仓库，这样其他人也能体验一下你的Node包。

首先我们需要在npm仓库上注册一个账号。我们可以运行npm adduser命令来完成注册，注册时会有一系列问题要回答并填写，以设置用户名、密码和电子邮件地址。

```
$ npm adduser
Username: my-user-name
Password:
Email: me@example.com
```


下一步我们要在对应Node包的根目录下运行npm publish命令：

```
$ npm publish
```

如果上面的操作一切正常，在npm publish执行结束之后，你可以访问http://search.npmjs.org，然后搜索Node包。搜索结果很快就会显示出来。

顾名思义，npm unpublish命令可以将Node包从npm仓库里移除。

11. npm 的配置

我们已经在之前全局模式和本地模式的比较中配置过npm，但还有一些其他的设置可调优npm的运行。让我们先看下配置方式。

首先是npm set 和npm get命令，具体配置方式为：

```
npm config set <key> <value> [--global]
npm config get <key>
npm config delete <key>
npm config list
npm config edit
npm get <key>
npm set <key> <value> [--global]
```

具体例子如下：

```
$ npm set color true
$ npm set global false
$ npm config get color
true
$ npm config get global
false
```

环境变量也可以用来配置npm。所有以NPM_CONFIG_开头的变量都是npm的配置项。例如，NPM_CONFIG_GLOBAL用于设置全局的配置global的值。

可以放到配置文件里的配置值：

- $HOME/.npmrc
- <Node Install Directory>/etc/npmrc

配置文件里的键值对如下，键值对会在npm config set命令执行后自动更新：

```
$ cat ~/.npmrc
global = false
color = true
```

3.2.4　Node 包版本的标识和范围

Node无法识别任何和版本号相关的内容。它能识别模块，而且会像对待一个模块一样对待目录结构，它具有功能相当丰富的模块查找系统，但是它无法识别版本号。然而，npm是可以识别版本号的。它使用了Semantic Versioning模式（在后面介绍），就像我们已经看到的，我们可以通过网络安装模块，查找旧版的模块并用npm对其进行更新。所有这些都属于版本控制的范畴，那么让我们仔细看看有了版本号和版本标签之后npm可以做的事情。

之前我们曾使用npm list命令列出已安装的包，列表包括了已安装包的版本号。如果你希望看到一个特定模块的版本号，可以输入下面的命令：

```
$ npm view express version
2.4.0
```

当npm命令参数中包含包名，你可以指定一个版本号或者版本标签。如果需要的话你可以通过这个方式处理特定的包版本。例如，如果你已经在测试环境对某个特定的版本完成了测试并证明其可用，那么就能保证这个版本能部署到生产环境中了：

```
$ npm install express@2.3.1
mime@1.2.2 ./node_modules/express/node_modules/mime
connect@1.5.1 ./node_modules/express/node_modules/connect
qs@0.2.0 ./node_modules/express/node_modules/qs
express@2.3.1 ./node_modules/express
```

npm有“标签”这个概念，我们可以利用它并用下面这样的命令安装最新版本的、稳定的Node包。

```
$ npm install sax@stable
```

标签名字可以是任意字符，而且不是必需的。包的作者可以指定标签名，且并不是所有的包都使用标签名。

Node包的package.json里会列出与其他包的依赖关系，你可以这样查看：

```
$ npm view mongoose dependencies
{ hooks: '0.1.9' }

$ npm view express dependencies
```

```
{ connect: '>= 1.5.1 < 2.0.0',
  mime: '>= 0.0.1',
  qs: '>= 0.0.6' }
```

npm需要按照包的依赖关系决定安装哪些模块。当安装一模块包的时候，它会查看声明的依赖关系，然后下载那些未安装的模块。

眼尖的读者可能已经看到这个例子里的小于号和大于号了。npm支持版本号范围的声明，比如当Express做了这个声明时，它在Connect 1.51 ~ 2.0.0版本下都可以正常运行。

npm背后有一个规则做指导，这可能对于一个一直和软件打交道的人来说很简单也很普通。npm作者使用http://semver.org上的Semantic Versioning标准作为npm版本号系统的指导标准，如下所示。

- 版本字符串通常都是*X.Y.Z*形式的整数，其中*X*是主版本，*Y*是副版本，*Z*是日常补丁的版本，例如1.2.3。
- 版本字符串的补丁号后可以有一个任意的文本，我们通常叫它“特别版本”，例如1.2.3beta1。
- 版本字符串的比较不是一般的字符串比较，而是一种数值比较，也就是比较*X*、*Y*、*Z*的值。例如，1.9.0 < 1.10.0 < 1.11.3，1.0.0beta1 < 1.0.0beta2 < 1.0.0。
- 用版本号来记录兼容性。
  - *X*值为0的包肯定是不稳定的，任何API都有可能随时改变。
  - 如果只是做了向前兼容的bug修复，*Z*的值（补丁号）必须递增。
  - 如果引入了向前兼容的函数，*Y*值必须递增（比如，一个新的函数引入，不过所有其他函数都还是兼容的）。
  - 当与之前版本有冲突的改变出现时，*X*的值就得递增。

3.2.5 CommonJS 模块

Node的模块系统是基于CommonJS（http://www.commonjs.org/）模块系统的。虽然JavaScript是一门强大的语言，含有很多高级的特性（比如对象和闭包），但是它缺少一个标准对象库来辅助构建应用。CommonJS旨在填补这个缺口，同时提供了一个实现JavaScript模块的约定和一系列标准模块。

`require`函数以模块标识符作为参数，返回模块对外开放的接口。如果要加载的模块又引用了其他模块，后者也要被加载。模块包括在一个JavaScript文件中，CommomJS并不指定模块标识符对应文件名的方式。

模块提供了一个简单的封装机制去隐藏本身的实现方式，而暴露出一个API。模块内容就是JavaScript代码，就像是以如下的方式编写：

```
(function() { ... contents of module file ... })();
```

这样的方式将模块的所有顶级对象隐藏在一个私有的命名空间中，使其他代码无法访问。这就是全局对象污染问题的解决方案（稍后详细介绍）。

模块对外开放的API即是require函数返回的对象。在模块内部，这个API是由exports对象实现的，exports的字段包含被暴露的API。为了从模块中暴露出一个函数或者对象，我们只需要将其注册到exports对象里。

模块封装演示

让我们看一个例子，先创建一个含有如下代码的module1.js：

```
var A = "value A";
var B = "value B";
exports.values = function() {
  return { A: A, B: B };
}
```

然后创建一个含有下面代码的module2.js：

```
var util = require('util');
var A = "a different value A";
var B = "a different value B";
var m1 = require('./module1');
util.log('A='+A+' B='+B+' values='+util.inspect(m1.values()));
```

在Node里运行：

```
$ node module2.js
19 May 21:36:30 - A=a different value A B=a different value B values={ A: 'value A',
B: 'value B' }
```

这个例子展示了模块内部值的封装方法，它使得module1.js里的A和B并不会覆盖module2.js里的A和B，因为它们是被封装在module1.js内的。模块内部经过封装的值也可以暴露给外部，比如module1.js里的.values函数。

之前提到的全局对象问题和全局环境内的变量有关。在Web浏览器里，有一个全局的执行上下文，如果一个JavaScript脚本修改了在其他脚本中使用的全局变量，这会造成很多问题。对于CommonJS模块，每一个模块都有自己的私有全局执行环境，这样模块内的多个函数共享变量也变得较为安全，不用担心会影响其他模块中的全局变量。

3.3　小结

这一章我们学习了很多关于Node模块和包的内容，具体内容包括：

- 实现Node模块和包；
- 管理安装的模块和包；
- Node如何定位模块。

学习了Node模块和包，接下来就该在构建应用的时候使用它们了，这也正是下一章要介绍的内容。

第 4 章

几种典型的简单应用

学习完Node模块之后，接下来就是学以致用的时候了。在这一章里，我们会尽量保持应用足够简单，从而可以把研究重点放在3个不同的Node应用框架上。之后几章我们会做一些复杂的应用，俗话说得好：学会跑之前，得先学会走。

开始吧。

4.1　Math Wizard

在这一章，我们所要做的就是一个简单的Math Wizard应用。Math Wizard的用户体验很不错，适合用于对儿童讲授算术。由于我们没有擅长用户体验的行家，所以做出来的Math Wizard应用只能用于对Node使用者讲授Web应用开发。有一点需要事先提醒下，不要期望你的孩子能够通过这个应用成为数学天才。

Math Wizard应用包括了主页、侧边导航栏和其他几个允许用户进行算术操作的页面。

是否使用 Web 框架

Web框架可以让你从复杂的HTTP协议实现中解脱出来。远离细节是一个程序员保持高效的工作方式。当你使用一个库或者框架，而这个库或者框架提供了很多封装好的函数来帮助处理细节时，这个感觉会特别明显。

在这一章，我们会先从编写一个不依赖于框架的应用（Math Wizard）开始，然后使用Connect和Express分别进行渐进增强式的开发。

4.2　不依赖框架的实现

我们先循序渐进学习Node应用的构建，慢慢感受Web框架带来的便捷。这意味着我们要从Node的核心包HTTP Server对象开始学起。

和其他Web应用一样，Math Wizard包含很多页面，每个页面对应一个URL。每个页面都有一些常见的元素（常用的页面结构和导航栏），不过每个页面的内容是不同的。在Math Wizard中，URL规则如下：

- / 　wizard应用的主页；
- /square　用于计算一个数的平方；
- /mult　用于计算两个数相乘；
- /factorial　用于计算一个数的阶乘；
- /fibonacci　用于计算斐波那契数。

我们先创建一个文件夹来存放源代码：

```
$ mkdir chap04
```

4.2.1　路由请求

Math Wizard的每一个页面都是由一个独立的模块实现，然后服务器把请求分别路由到这些模块。

我们所指的“请求的路由选择”就是一种把应用切分成多个模块的方式。与其在一个庞大的回调函数中实现应用的每一个细节，不如使用模块化的处理方式。请求的路由选择首先需要有代码对进来的HTTP请求进行检查，然后调用对应的模块处理请求。

创建一个app-node.js，其所包含内容如下：

```
var http_port = 8124;

var http    = require('http');
var htutil  = require('./htutil');

var server = http.createServer(function (req, res) {
  htutil.loadParams(req, res, undefined);
  if (req.requrl.pathname === '/') {
    require('./home-node').get(req, res);
  } else if (req.requrl.pathname === '/square') {
    require('./square-node').get(req, res);
  } else if (req.requrl.pathname === '/factorial') {
    require('./factorial-node').get(req, res);
  } else if (req.requrl.pathname === '/fibonacci') {
    require('./fibo-node').get(req, res);
    // require('./fibo2-node').get(req, res);
  } else if (req.requrl.pathname === '/mult') {
    require('./mult-node').get(req, res);
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("bad URL "+ req.url);
  }
});

server.listen(http_port);
console.log('listening to http://localhost:8124');
```

这个请求路由器是非常简单的。每一个HTTP请求的回调函数都使用了req变量来持有请求的数据，res用来持有响应数据。这一请求路由器会检查请求的URL，然后将请求转发到对应的处理函数。

这一应用由一些模块组成，这些模块分别对应一个页面且都暴露了一个叫做.get的函数。这个函数使用了function(req, res)语句来实现Math Wizard中的一个页面。

如果请求的URL没有匹配任何模块，我们会返回一个404状态码，这个状态码说明对应的页面没有找到。

4.2.2 处理 URL 查询参数

htutil.loadParams函数可以帮助我们完成URL的解析并保存解析后的对象，这样Math Wizard应用的其他部分也可以引用这个对象。每一个页面都会包括一个输入为a和b的表单。当用户输入数字并点击提交按钮Submit时，URL会包含一个查询字符串，如下所示：

```
http://localhost:8124/mult?a=3&b=7
```

这些查询参数只会在输入数字并点击提交按钮的时候生成。这意味着每一个Math Wizard页面必须同时兼容参数存在和不存在的情况。htutil.loadParams函数可以很方便地查找这些参数，从而减少各个模块中重复的代码。

创建一个htutil.js，使其包含如下的代码：

```
var url = require('url');
exports.loadParams = function(req, res, next) {
  req.requrl = url.parse(req.url, true);
  req.a = (req.requrl.query.a && !isNaN(req.requrl.query.a))
    ? new Number(req.requrl.query.a)
    : NaN;
  req.b = (req.requrl.query.b && !isNaN(req.requrl.query.b))
    ? new Number(req.requrl.query.b)
    : NaN
    if (next) next();
}
```

4

loadParams函数会由HTTP请求处理函数调用，并接收HTTP请求处理函数传递的req和res参数。它会查找a和b对应的查询参数，并将其添加到req对象里。例子中我们使用:?运算符来确保req.a和req.b拥有值NaN或者一个数字，这取决于查询参数是否存在，以精简其他代码。现在你可以先忽略名为next的函数，我们会在后面介绍Connect框架时讨论这个方法。

在htutil.js里还有另外两个函数，它们负责页面布局的处理。Math Wizard给每个页面提供了一个通用的布局，布局代码的集中处理可以减少重复的代码。在稍后使用Express框架的时候，我们可以使用模板文件来控制页面布局，但是现在在这一版Math Wizard中我们只使用Node的核心部分，自然也就不使用模板了。

在htutil.js里，我们继续添加下面两个函数。它们是两个帮助构造页面的效用函数。

```
exports.navbar = function() {
    return ["<div class='navbar'>",
        "<p><a href='/'>home</a></p>",
        "<p><a href='/mult'>Multiplication</a></p>",
        "<p><a href='/square'>Square's</a></p>",
        "<p><a href='/factorial'>Factorial's</a></p>",
```

```
        "<p><a href='/fibonacci'>Fibonacci's</a></p>",
        "</div>"].join('\n');
}
```

这个函数提供了用于链接其他页面的HTML片段。它可以作为一个导航栏，使用户通过它访问应用的各个页面。

```
exports.page = function(title, navbar, content) {
    return ["<html><head><title>{title}</title></head>",
      "<body><h1>{title}</h1>",
      "<table><tr>",
      "<td>{navbar}</td><td>{content}</td>",
      "</tr></table></body></html>"
      ].join('\n')
      .replace("{title}", title, "g")
      .replace("{navbar}", navbar, "g")
      .replace("{content}", content, "g");
}
```

这个函数对应的是整个页面的HTML结构。它调用了一些参数来填充标题、导航栏和页面的内容。

我们在这里使用了正则表达式和`replace`函数，从而可以很方便地将数据替换为字符串。`replace`函数是一个字符串函数，它使用了一个正则表达式来匹配字符串，然后用提供的字符串替换匹配成功的文本。

下一节我们会介绍如何使用这些函数。

4.2.3　乘法运算

现在让我们看下如何在Math Wizard里创建一些能够进行数学运算的页面。首先要实现的是乘法（比如`a*b`）页面。

创建一个`mult-node.js`，使其包含下面的代码：

```
var htutil = require('./htutil');
exports.get = function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  var result = req.a * req.b;
  res.end(
    htutil.page("Multiplication", htutil.navbar(), [
      (!isNaN(req.a) && !isNaN(req.b) ?
        ("<p class='result'>{a} * {b} = {result}</p>"
          .replace("{a}", req.a)
          .replace("{b}", req.b)
          .replace("{result}", req.a * req.b))
          : ""),
      "<p>Enter numbers to multiply</p>",
      "<form name='mult' action='/mult' method='get'>",
      "A: <input type='text' name='a' /><br/>",
      "B: <input type='text' name='b' />",
```

```
        "<input type='submit' value='Submit' />",
        "</form>"
      ].join('\n'))
    );
}
```

和其他Math Wizard模块一样，乘法模块有两个目的。首先是显示算术结果，然后是显示一个让用户输入数字（一个或两个）的表单，用于乘法运算。

首先要注意的是我们使用了`htutil.page`函数。它提供了总体的页面布局控制功能，在这个函数里，我们只提供页面的主要内容区域。这个内容区域是一个字符串数组，最后会被用`.join()`函数联接起来。

如果用户提供了一个参数，那么最关键的部分就是按照下面的代码显示结果：

```
(!isNaN(req.a) && !isNaN(req.b) ?
  ("<p class='result'>{a} * {b} = {result}</p>"
    .replace("{a}", req.a)
    .replace("{b}", req.b)
    .replace("{result}", req.a * req.b))
: ""),
```

这个例子使用了`?:`运算符来检查参数是否已经提供，如果提供了，那么对`req.a`和`req.b`进行乘法运算并显示结果。

4.2.4　其他数学函数的执行

其他的Math Wizard模块和`mult-node.js`类似，使用一样的模式创建，让我们快速了解一下。

一个数的平方是这个数乘以它本身（比如`a*a`）。创建`square-node.js`，使其包含下面的代码。注意，我们使用了`Math.floor`方法来确保对`req.a`的值取整。

```
var htutil = require('./htutil');
exports.get = function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.end(
    htutil.page("Square", htutil.navbar(), [
      (!isNaN(req.a) ?
        ("<p class='result'>{a} squared = {sq}</p>"
          .replace("{a}", req.a)
          .replace("{sq}", req.a*req.a))
      : ""),
      "<p>Enter a number to see its square</p>",
      "<form name='square' action='/square' method='get'>",
      "A: <input type='text' name='a' />",
      "</form>"
    ].join('\n'))
  );
}
```

`n`的阶乘，用数学表达式表示是`n!`。它是`n`及所有小于`n`的正整数相乘的结果。阶乘运算在数

4

学的很多领域都会用到。接下来创建一个factorial-node.js，使其包含下面的代码：

```
var htutil = require('./htutil');
var math  = require('./math');

exports.get = function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.end(
    htutil.page("Factorial", htutil.navbar(), [
      (!isNaN(req.a) ?
        ("<p class='result'>{a} factorial = {fact}</p>"
        .replace("{a}", req.a)
        .replace("{fact}",
          math.factorial(Math.floor(req.a))))
      : ""),
      "<p>Enter a number to see it's factorial</p>",
      "<form name='factorial' action='/factorial' method='get'>",
      "A: <input type='text' name='a' />",
      "</form>"
    ].join('\n'))
  );
}
```

斐波那契数就是如下顺序的一组整数：0、1、1、2、3、5、8、13、21、34、55等。斐波那契数列中的每个数都是它前两个数之和。连续数的比例接近于黄金比例。让我们创建一个名为fibo-node.js的文件，用于创建计算斐波那契数的页面：

```
var htutil = require('./htutil');
var math  = require('./math');
exports.get = function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.end(
    htutil.page("Fibonacci", htutil.navbar(), [
      (!isNaN(req.a) ?
        ("<p class='result'>fibonacci {a} = {fibo}</p>"
          .replace("{a}", Math.floor(req.a))
        .replace("{fibo}", math.fibonacci(Math.floor(req.a))))
       : ""),
      "<p>Enter a number to see its fibonacci</p>",
      "<form name='fibonacci' action='/fibonacci' method='get'>",
      "A: <input type='text' name='a' />",
      "</form>"
    ].join('\n'))
  );
}
```

一些眼尖的读者可能已经注意到一个叫做math的模块。这个模块包含了一些数学函数的实现。我们需要创建一个math.js文件，使其包含下面的代码：

```
var factorial = exports.factorial = function(n) {
```

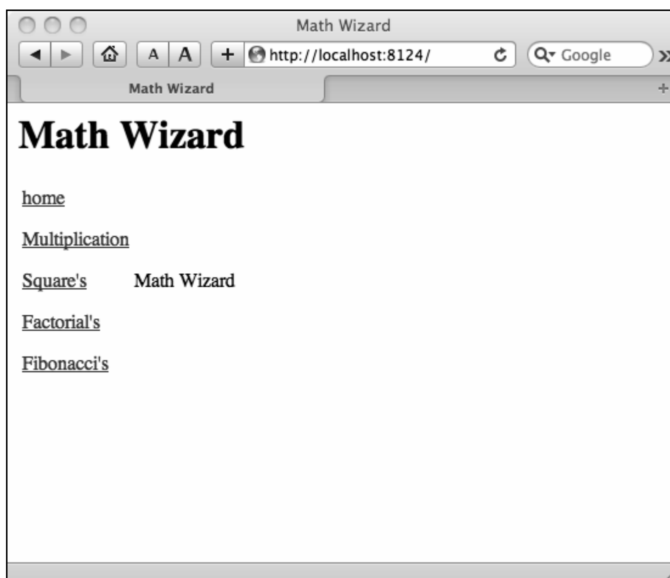
```
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}

var fibonacci = exports.fibonacci = function(n) {
  if (n === 1)
    return 1;
  else if (n === 2)
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

这些方法简单地实现了一些标准的数学函数。就像我们能马上看到的，斐波那契函数是一个非常简单的计算密集型函数。

我们希望Math Wizard应用能有一个主页。创建一个`homenode.js`，使其包含下面的代码：

```
var htutil = require('./htutil');
exports.get = function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.end(
    htutil.page("Math Wizard",
      htutil.navbar(),
      "<p>Math Wizard</p>")
  );
}
```



输入下面的命令：

```
$ node app-node.js
```

因为app-node.js监听了8124端口，所以访问http://localhost:8124/的时候我们可以看到下面的内容。

4.2.5 扩展 Math Wizard

我们的孩子需要高质量的教育，而这个示例程序只是可教授算术的基本应用，但也许并不适合。数学运算是多种多样的，因此任何情况下，我们都可以很轻易地扩展Math Wizard，为其添加其他页面。遵循已有的模式给Math Wizard添加一个新的页面需要用到下面几个步骤。

- 在htutil.navbar上添加一个a标签。

  因为htutil.navbar函数包含了针对导航栏的HTML代码，任意一个新的Math Wizard页面都需要被像下面这样列出页面的URL和页面的名字：

```
"<p><a href='/newUrl'>Math Function Name</a></p>\n"+
```

- 在app-node.js里增加一条if语句。

  因为app-node.js文件包含请求的路由功能，所以需要一个新的if语句实现我们刚刚定义的URL的转发。这个URL需要匹配htutil.navbar函数里的URL。

```
if (req.requrl.pathname === '/newUrl') {
        require('./moduleName').get(req, res);
    }
```

- 增加一个页面处理函数模块，对外暴露一个get方法。

  我们之前已经看到一些处理函数模块（mult-node.js等），按照那些例子再创建一个这种模块还是很容易的。

4.2.6 长时间运行的运算（斐波那契数）

应用Math Wizard展示了Node应用中一个饱受批评的缺点。如果回调函数不能对请求进行快速的处理并返回Node事件循环，应用就会陷入泥潭。

为了让大家更直观地了解这个缺点，我们可以在斐波那契数计算页面上输入一个很大的数，比如50。这样就需要大量的时间来执行运算（大概需要几个小时到几天），Node进程的CPU占用率会变高，然后在其他浏览器窗口内，你可能无法使用Math Wizard了。所有这一切都是因为斐波那契序列的数的计算是非常庞大的任务。为什么浏览器也会无响应？这是因为密集型计算阻止了Node事件循环的执行，从而阻止了Node对浏览器请求进行的响应。

由于Node只有一个执行线程，请求处理程序必须在执行完之后快速返回事件循环。通常，异步编程能保证事件循环正常执行。甚至在加载地球另一端的服务器数据时，异步编程的作用也能体现出来，因为I/O是非阻塞的，控制能很快返回给事件循环。因为我们选择的简单斐波那契函数是一个长时间运行的阻塞型计算，所以它并不适用于这一模型。这样的事件处理程序会让系统

不用忙于处理请求，并让Node从即将处理的事务中解脱出来，从而实现一个极快的Web服务器。

在这个例子里，长时间运行的计算导致的问题（长响应时间问题）很明显。用于响应斐波那契数的时间几乎让你有时间去西藏进行一次欢快的旅行。长响应时间可能在你的应用中不是很明显，那么你如何知道自己的请求会占用太多时间呢？一个测量手段就是利用浏览器工具（比如YSlow）显示的响应延迟。用户使用浏览器的时候，能在一两秒内显示下一页是很重要的，否则你将有可能失去你的用户。

在Node里有两种解决这类问题的常见方案。

- **算法重构** 就像我们选择的斐波那契函数一样，非最佳的算法可以通过重写实现更高的效率。或者，即使没有优化空间，我们也可将其切分并通过事件循环分派到不同回调函数里。稍后我们会具体介绍一下这种方法。
- **创建一个后台服务** 你能想象一个专门用于计算斐波那契数的后台服务器吗？好吧，或许不能，但是这确实是非常普遍的将前台服务转移到后台执行的方法，我们会在这章的最后介绍如何实现一个后台运行的数学计算服务器。这个请求处理函数应可以异步地调用数据服务或者数据库、收集响应数据，然后发送给浏览器。

我们可以对斐波那契算法进行优化，但我们不这样做，而是将其从一个同步的函数转换成一个含有对应回调的异步函数。在这里使用异步的斐波那契算法并不是一个很好的选择，但它的确展示了算法重构方法。我们选择了对计算过程进行拆分，并通过事件循环分派给对应的回调函数。

首先，我们要对斐波那契函数进行实例化，用实例化后的对象取代原来实现的函数。如果已经写了一个比较原始又低效的函数，之后你可能不得不用一个更好的替换它。接下来在math.js中添加如下代码：

```
var fibonacciAsync = exports.fibonacciAsync = function(n, done) {
  if (n === 1 || n === 2)
    done(1);
  else {
    process.nextTick(function() {
      fibonacciAsync(n-1, function(val1) {
        process.nextTick(function() {
          fibonacciAsync(n-2, function(val2) {
            done(val1+val2);
          });
        });
      });
    });
  }
}
```

这就是我们的新的异步斐波那契算法。我们已经把它从一个简单的函数转换成一个异步驱动的计算函数，这样就可以通过回调函数传递结果，如下所示：

```
fibonacciAsync(n, function(value) {
   // 操作值
});
```

我们使用了process.nextTick方法将一个递归函数转换成各个步骤都由事件循环分派。这

个函数通过事件循环调用回调函数，确保函数能快速进入事件循环，这样服务器才能继续处理别的HTTP请求。这不是唯一一个通过事件循环派发算法步骤的方式。异步模块也可以做这个，它有一系列方法帮助实现异步的JavaScript。

在fibonacciAsync函数里，process.nextTick方法替换了原始算法里的这一表达式：

```
return fibonacci(n-1)+fibonacci(n-2);
```

这句代码的任务是计算两个斐波那契数，然后对其执行加法，最后将结果返回给调用函数。新的算法有3个异步函数来实现任务的各个步骤，并使用了process.nextTick方法确保这些都通过事件循环分派执行。

在我们介绍后面的内容之前，先花点时间斟酌下这个方案。它没有减少必需的计算量，只是将计算过程交给事件循环调度。它会使当前的Node进程占用所有CPU负载，这绝不是重构计算密集型算法（如斐波那契算法）的最佳途径。但是这展示了通过事件循环分派工作的技术，这个技术对一些算法来说是非常实用的，但是对其他算法就并不是那么实用。

这个技术是否实用还是取决于你和你使用的算法，你需要选择最佳的方法来处理长时间运行的计算。例如，这章的后面部分，我们会展示如何实现一个可以用HTTP访问的后台服务器，这项技术可以将计算任务推送到任何模块。

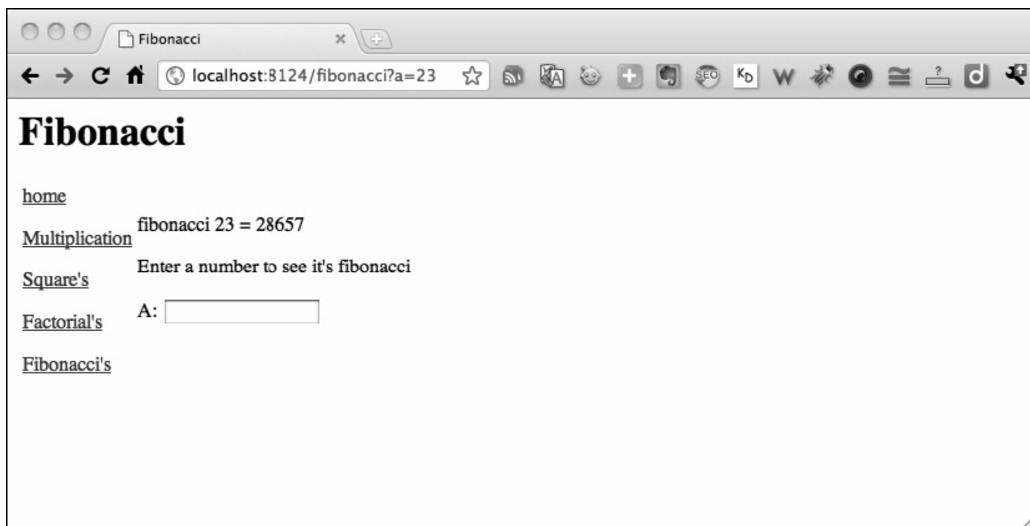
创建一个新文件fibo2-node.js，然后将app-node.js修改为require('./fibo2-node')，这样就可以使用新的斐波那契模块了。我们已经将这行代码添加到app-node.js文件里，不过它被注释掉了。接下来我们可以切换代码的注释来改变斐波那契算法的实现：

```
var htutil = require('./htutil');
var math  = require('./math');
function sendResult(req, res, a, fiboval) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.end(
    htutil.page("Fibonacci", htutil.navbar(), [
      (!isNaN(fiboval) ?
        ("<p class='result'>fibonacci {a} = {fibo}</p>"
          .replace("{a}", a)
          .replace("{fibo}", fiboval))
      : ""),
      "<p>Enter a number to see its fibonacci</p>",
      "<form name='fibonacci' action='/fibonacci' method='get'>",
      "A: <input type='text' name='a' />",
      "</form>"
    ].join('\n'))
  );
}

exports.get = function(req, res) {
  if (!isNaN(req.a)) {
    math.fibonacciAsync(Math.floor(req.a), function(val) {
      sendResult(req, res, Math.floor(req.a), val);
    });
```

```
    } else {
      sendResult(req, res, NaN, NaN);
    }
  }
```

我们已经通过将运算转移到sendResult函数重构了斐波那契模块，我们以两种方式调用send Result，并依据是否需要显示斐波那契数判断以哪种方式调用。



虽然请求较大的斐波那契数依然需要很长的时间来计算，但是服务器不会阻塞，可以处理其他请求。当打开多个浏览器标签页的时候，你就会明显感觉到这点。在一个标签页中请求一个大的斐波那契数会需要很长的时间来计算。然后在另一个标签页中发起一个请求，就像你在截图里看到的，浏览器不是处于无响应状态，而是在顺利地处理请求。

4.2.7　还缺什么功能

在后面讨论Connect时，我们会发现Math Wizard里极小的dispatch函数做的事情和实际Web服务器做的相比要少一些。仅仅实现HTTP协议并不能造就一个完善的Web服务器或者Web应用，因为它缺失了很多近20年Web应用开发中积累下的最佳实践。

- Math Wizard不能关注请求的方式（GET、PUT、POST等）。要维持HTTP的语义就需要对GET、PUT或者POST请求进行不同的处理。
- 没有给错误的URL请求提供404页面。
- 没有为URL和表单屏蔽注入式的脚本攻击。
- 不支持cookie处理，也没有使用cookie维持会话。
- 不记录请求。
- 不支持身份验证。
- 不处理图像、CSS、JavaScript或者HTML这样的静态文件。

- 没有对页面尺寸、执行时间等施加限制。

就像我们将在介绍Connect和Express时看到的那样，Node的Web框架弥补了大部分丢失的特性。

4.2.8　使用 Connect 框架实现 Math Wizard

Connect（http://senchalabs.github.com/connect/）其实不是一个真正意义上的Web框架，而是Node的一个中间件[1]框架。它含有11个已绑定的中间件并具有丰富的第三方中间件以供选择。“中间件”这个术语或许会因为太过笼统让你感到困惑，因此接下来我们对这个词进行详细介绍。

TJ Holowaychuck将“中间件”描述为易于挂载和调用的模块，可以“无序”使用并为应用的快速开发提供常用的功能，比如请求的路由选择、身份验证、请求记录、cookie处理等（详见http://tjholowaychuk.com/post/664516126/connect-middleware-for-nodejs）。

中间件有以下两种形式。

- **过滤器**　处在中间层负责处理传入和传出的请求，但是本身不响应请求。过滤器的一个例子就是中间件*logger*，它专门提供可定制的信息记录。
- **供应者**　作为堆栈的终端，意味着一个进入的请求最后会在供应者处处理，并且供应者负责发送响应。供应者的一个例子就是为静态文件服务的中间件*static*。

在前面一节，我们看到了一个以`http.createServer`对象和一个函数构建的应用，这个函数会在每一个HTTP请求到达服务器的时候被调用。在使用Connect的时候，你就只需要用到`connect.createServer`对象，然后将中间件模块绑定到服务器上。作为中间件模块之一，`router`模块用于实现应用的URL。

了解这些内容之后，我们可以开始看一些代码了。

4.2.9　安装和设置 Connect

首先，确保Connect已经安装好了：

```
$ npm install connect
```

然后创建`app-connect.js`文件，使其内容如下：

```
var connect = require('connect');
var htutil  = require('./htutil');

connect.createServer()
  .use(connect.favicon())
  .use(connect.logger())
  .use('/filez', connect.static(__dirname + '/filez'))
  .use(connect.router(function(app){
    app.get('/',
      require('./home-node').get);
```

---

① 中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。

```
    app.get('/square', htutil.loadParams,
      require('./square-node').get);
    app.get('/factorial', htutil.loadParams,
      require('./factorial-node').get);
    app.get('/fibonacci', htutil.loadParams,
      require('./fibo2-node').get);
    app.get('/mult', htutil.loadParams,
      require('./mult-node').get);
  })).listen(8124);
console.log('listening to http://localhost:8124');
```

然后按照下面的方式启动服务器：

```
$ node app-connect.js
```

因为服务器监听了8124端口，所以直接在浏览器里访问http://localhost:8124/即可使用它。

恭喜！我们现在已经成功运行了第一个基于Connect开发的Node应用。

你会发现它看起来（及其行为）和之前的Math Wizard很类似，这是因为app-connect.js模块重用了app-node.js的模块。app.get函数所做的只是把请求转发到一个已存在的模块。

如果行为类似的话，那么Connect带来的优势在哪里呢?

不同点在于Connect提供了一个请求处理和调度框架来减轻应用开发的压力。它会帮你处理之前缺失的作为一个“完备Web服务器”应该有的特性，让你把精力更多的放在自己的应用上。但是这足以让Connect成为一个应用框架吗?

Connect没有以一个应用框架的形式呈现出来，而是作为一个应用框架的构建基石存在。Express就是一个建立在Connect之上的框架。Connect本身是非常实用的，了解Connect有助于理解Express，所以接下来我们会简单介绍下Connect，然后再介绍Express。

4.2.10 使用 Connect

我们已经学习了一部分有关Connect的知识，接下来看一些更详细的内容。Connect是Express框架的基础，它几乎突破了所有之前讨论过的基于HTTP服务器对象构建应用的限制。不要着急，我们先看一下app-connect.js。

在Connect里有很多配置服务器对象的方式。我们在app-connect.js里是这么做的：

```
var connect = require('connect');
connect.createServer()
  .use(connect.favicon())
  .use(connect.logger())
  .use('/filez', connect.static(__dirname + '/filez'))
  .use(connect.router(function(app){
    // 配置路由器
  })).listen( .. port number ..);
```

.use方法可用于绑定中间件到Connect服务器。它会配置一系列在接到请求时调用的中间件模块。当然，你的应用决定了使用哪些中间件模块。

.use方法允许我们链式调用.use从而获得更好的编程体验(server.use().use().use().use())。

在这个例子里我们要配置的中间件有*favicon*、*logger*、*static*和*router*。

在创建一个Apache式访问记录时，logger中间件非常有用。默认情况下，logger会将数据输出到控制台上，但是也可以通过配置以其他任何格式输出或者记录到任意文件里。

static中间件实现一个“静态Web服务器”来传递存放在特定目录下的文件。这意味着如果你有一个目录，其中包含了要发送给浏览器的.html，.css或者.js文件，connect.static中间件可以帮你完成任务。

favicon是一些浏览器在地址栏、标签栏显示的小图片，这些小图片也是一个你的品牌存在的标志，favicon中间件会帮助处理这些图片。

router中间件用于将每一个URL请求正确传递到对应的处理函数。配置方法如下：

```
.use(connect.router(function(app){
  // 配置路由器
})
```

但是真正占主导地位的是路由器的配置代码，在配置代码处你可以声明应用要识别的URL，还有每一个URL对应的处理函数。router的配置形式如下：

```
app.requestName('path', function(req, res, next) {..});
```

请求名指的是指HTTP动作，比如get、put、post等。这意味着你可以在页面上设置一个HTML表单，将method设置为POST，然后用app.get函数将页面传递给浏览器，然后使用app.post函数来接收表单传递回的数据。我们会在第6章看到对应的例子，不过它看起来更像下面定义的函数：

```
app.get('/form', createPageWithForm);
app.post('/form', receiveValuesPostedWithForm);
```

上面例子里的回调函数比一般的请求处理函数多了一个函数类型的参数。它的定义形式是function(req, res, next)，req和res分别对应了HTTP请求数据和HTTP响应数据。next这个参数对应的是Connect提供的函数，用于确保所有的中间件函数已经执行。

就像我们在app-connect.js里做的那样，路由可以分配到多个函数中。只要使用了next函数，Connect就可以依次对函数进行调用。和app-node.js里一样，我们在app-connect.js里使用的也是htutil.loadParams函数。你应该还记得，它使用了Connect提供的next函数。

这是一个典型的路由器配置函数：

```
app.get('/square', htutil.loadParams,
 require('./square-node').get);
```

这个函数的参数是一个URL字符串和两个函数类型的参数，第一个是htutil.loadParams函数。路由器配置函数可以包含不限数量的函数，你可以为自己的应用构造一个处理函数队列。

总而言之，中间件和多个路由器函数组成了一种处理HTTP请求的状态机。我们已经看到了两个函数列。第一个是列在服务器配置中的中间件函数，还有就是我们刚刚看到的路由器函数列表。

4.3 使用 Express 框架实现 Math Wizard

现在我们已经对Connect有所了解，接下来用Express对Math Wizard进行改进。Express是一个基于Connect（一个中间件框架）的Web应用框架。这意味着Express专注于构建一个应用，包括提供一个模板系统，而Connect专注于做Web服务的基础设施。Express和Connect是由同一个团队开发的，所以它们的API相似度比较高。

例如，这是用Express实现的一个Hello World示例：

```
var app = require('express').createServer();
app.get('/', function(req, res) {
  res.send('Hello, world!');
});
app.listen(3000);
```

这个例子看起来和前面一节的代码比较类似。然而，`createServer`返回的对象中包含被绑定的路由器中间件函数。看起来你似乎跳过了大部分中间件的绑定和配置，直接使用了URL路由器。当然，你还可以继续绑定和配置中间件：

```
var express = require('express');
var app = express.createServer(
  express.logger(),
  express.bodyParser()
);
```

安装Express和EJS（模板处理系统）只需要做下面的操作：

```
$ npm install express ejs
qs@0.1.0 ../node_modules/express/node_modules/qs
express@2.3.11 ../node_modules/express
ejs@0.4.2 ../node_modules/ejs
```


4.3.1 准备工作

必需的模块安装好后，我们可以写代码了。不过需要先做点准备工作，创建一个目录：

```
$ mkdir views
```

然后创建`app-express.js`，使其包含下面的代码：

```
var htutil  = require('./htutil');
var math  = require('./math');
var express = require('express');
var app = express.createServer(
  express.logger()
);

app.register('.html', require('ejs'));
// 可选，因为 Express 下默认为CWD/views
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

```
app.configure(function(){
  app.use(app.router);
  app.use(express.static(__dirname + '/filez'));
  app.use(express.errorHandler({
    dumpExceptions: true, showStack: true }));
});
```

我们启动了服务器，然后配置了必需的中间件。细节上会有一些区别，例如我们使用了express.logger而不是connect.logger，不过看起来还是很类似。

新用到的方法有app.register和app.set。因为这里展示的配置项对应的是模板系统的配置，所以.html文件会由EJS引擎处理。过会儿我们会看到res.render如何通过模板引擎将数据呈现到模板里。

现在看下路由器配置（依旧在app-express.js中进行）：

```
app.get('/', function(req, res) {
  res.render('home.html', { title: "Math Wizard" });
});
app.get('/mult', htutil.loadParams, function(req, res) {
  if (req.a && req.b) req.result = req.a * req.b;
  res.render('mult.html', {
    title: "Math Wizard" , req: req });
});
app.get('/square', htutil.loadParams, function(req, res) {
  if (req.a) req.result = req.a * req.a;
  res.render('square.html', {
    title: "Math Wizard" , req: req });
});
app.get('/fibonacci', htutil.loadParams, function(req, res) {
  if (req.a) {
    math.fibonacciAsync(Math.floor(req.a), function(val) {
      req.result = val;
      res.render('fibo.html', {
        title: "Math Wizard" , req: req });
    });
  } else {
    res.render('fibo.html', {
      title: "Math Wizard" , req: req });
  }
});
app.get('/factorial', htutil.loadParams, function(req, res) {
  if (req.a) req.result = math.factorial(req.a);
  res.render('factorial.html', {
    title: "Math Wizard" , req: req });
});

app.get('/404', function(req, res) {
  res.send('NOT FOUND '+req.url);
});

app.listen(8124);
console.log('listening to http://localhost:8124');
```

除了两者毫不相干的部分之外，Express里的路由器配置很大程度上和Connect一样。就像之前提示过的，路由器函数可以直接作用于服务器对象。最主要的不同点在于我们在路由器函数里依赖了模板引擎的部分。

在Express里，我们使用res.render函数而不是res.writeHead或者res.end来发送页面。res.render函数通过一个模板文件渲染数据，让我们能更好地实现数据和表现的分离。

EJS只是Express里众多模板引擎中的一个。我们的配置目的就是让EJS能够为views目录下的所有.html文件服务。

Express里还有其他一些模板引擎，而文件后缀可以用于指定对应的模板引擎，如下所示：

```
res.render('index.haml', {..data..}); // 使用Haml
res.render('index.jade', {..data..}); // 使用Jade
res.render('index.ejs', {..data..}); // 使用EJS
res.render('index.coffee', {..data..}); // 使用CoffeeKup
res.render('index.jqtpl', {..data..}); // 使用jQueryTemplates
```

你也可以通过app.set方法改变默认的渲染引擎：

```
app.set('view engine', 'haml'); // 使用Haml
app.set('view engine', 'jade'); // 使用Jade
app.set('view engine', 'ejs'); // 使用EJS
```

我们已经讨论过代码的具体实现，接下来可以创建模板文件了。所有的模板文件都要放到views目录下。

首先，在layout.html里，我们添加如下代码：

```
<html>
  <head><title><%= title %></title></head>
  <body>
    <h1><%= title %></h1>
    <table>
      <tr><td>
          <div class='navbar'>
            <p><a href='/'>home</a></p>
            <p><a href='/mult'>Multiplication</a></p>
            <p><a href='/square'>Square's</a></p>
            <p><a href='/factorial'>Factorial's</a></p>
            <p><a href='/fibonacci'>Fibonacci's</a></p>
          </div>
        </td>
        <td><%- body %></td>
      </tr>
    </table></body></html>
```

Express里的layout模板比较特殊。注意，在app.js里我们使用了res.render('fibo.html'..)而没有提到layout.html。这是为什么呢？默认情况下，模板中用于渲染的内容会被命名为body，然后传递到layout模板中。当app.js调用res.render('fibo.html'..)时，它会先用fibo.html渲染对应的页面片段，然后再使用layout模板渲染整个页面。

现在有两种方法覆盖这一默认的行为。第一种是在Express里创建一个全局的配置，通过这个全局配置来控制layout模板的启用与禁用。

```
app.set('view options', { layout: false (or true) });
```

第二种方法是覆盖layout模板对应的渲染方式：

```
res.render('myview.ejs', { layout: false (or true) });
```

我们可以在一个特定的渲染过程中关闭或者启用layout模板，或者通过下述代码使用不同的layout模板：

```
res.render('page', { layout: 'mylayout.jade' });
```

EJS模板大部分基于HTML规范，而3种特别的标签除外。如果你使用过其他模板系统，应该很熟悉这些标签，如下所示：

- 用`<% code %>`表示用于控制逻辑的不被缓存的代码；
- 用`<%= code %>`表示使用默认转义的html代码；
- 用 `<%- code %>` 表示未转义的被缓存代码。

我们可以在例子里看到使用`<%= title %>`标签代表已转义的HTML，使用`<%- body %>`标签表示的不需要缓存的数据。

接下来让我们回到Math Wizard模板home.html：

```
<p>Math Wizard</p>
```

home.html模板含有的内容就是上面这部分。

在mult.html里我们添加下面的代码：

```
<% if (req.a && req.b) { %>
  <p class='result'>
    <%= req.a %> * <%= req.b %> = <%= req.result %>
  </p>
<% } %>
<p>Enter numbers to multiply</p>
<form name='mult' action='/mult' method='get'>
  A: <input type='text' name='a' /><br/>
  B: <input type='text' name='b' />
  <input type='submit' value='Submit' />
</form>
```

在这里我们使用`<% code %>`来介绍通过条件语句控制渲染的方式：

```
<% if (req.a && req.b) { %>
  conditional content
<% } %>
```

`<% code %>`标签里的代码就是JavaScript代码，可以做任何JavaScript可以做的事，不过在这个例子里我们只是在条件满足的情况下渲染一些内容。如果要使用一个数据列表或者数组，你可以使用while语句遍历和渲染数据项。

现在，我们在square.html里添加下面的代码：

```
<% if (req.a) { %>
  <p class='result'>
   <%= req.a %> squared = <%= req.result %>
```

```
  </p>
<% } %>
<p>Enter numbers to multiply</p>
<form name='square' action='/square' method='get'>
  A: <input type='text' name='a' />
  <input type='submit' value='Submit' />
</form>
```

现在在factorial.html里添加下面的代码：

```
<% if (req.a) { %>
 <p class='result'>
    <%= req.a %> factorial = <%= req.result %>
 </p>
<% } %>
<p>Enter a number to see it's factorial</p>
<form name='factorial' action='/factorial' method='get'>
  A: <input type='text' name='a' />
  <input type='submit' value='Submit' />
</form>
```

最后，在fibo.html里添加下面的代码：

```
<% if (req.a) { %>
  <p class='result'>
    fibonacci <%= req.a %> = <%= req.result %>
  </p>
<% } %>
<p>Enter a number to see it's fibonacci</p>
<form name='fibonacci' action='/fibonacci' method='get'>
  A: <input type='text' name='a' />
  <input type='submit' value='Submit' />
</form>
```


现在，一切都准备就绪，你可以使用下面的命令运行你的应用了：

```
$ node app-express.js
```

现在请用浏览器访问http://localhost:8124/，享受成果吧。

4.3.2 处理错误

错误是不可避免的。因为提前对错误进行检测会减少修复错误的工作量，所以我们最好能事先对错误有所了解。Express提供了两种捕获错误的方式。

在Math Wizard应用里我们有这样一行代码：

```
app.use(express.errorHandler({
  dumpExceptions: true, showStack: true
}));
```

这是默认的错误处理函数，展示了一个对高手和普通开发者都很友好的栈轨迹。你不会希望把这些内容直接展示给用户的。取而代之的是，你会想使用类似一群鸟把鲸鱼拖出水面那样的内容来展示。要展示用户友好的错误，首先要使用app.error函数安装一个错误事件处理函数。注

意，它所调用的函数有一个可选的参数err，用于存放错误对象。

```
app.error(function(err, req, res, next) {
// ……
  res.send(... error page); // or res.render('template'..)
});
```

一个有趣的错误页面可以很好地展示你的才华，当然你也可以创建一个无趣且乏味的错误页面。这些都取决于你自己。

4.3.3　参数化的 URL 和数据服务

到现在为止，我们已经探索过能发送HTML到浏览器的应用。虽然这是一个比较重要的使用场景，但是Express（及Connect）还可以用来做更多的事情。例如，我们通常使用HTTP构建REST服务，用于发送数据以供其他应用使用，而不是发送给用户HTML。

再早一点的时候，我们考虑（并排除）了将斐波那契计算从前台服务器转移到后台的可能性。接下来继续我们的学习并再构建一个应用，来看如何实现这个想法。同时，我们也会关注下Express的参数化路由选择和格式化响应数据的特性。让我们开始吧。

1. Express里参数化的URL

Express里的路由选择系统允许你在URL中指定req对象可以识别的占位符。这样你的应用会比没有参数化URL的应用更加灵活。整个过程是通过一组带URL标识符的类型匹配来完成的。Express会检查请求的URL，进行类型匹配，取出匹配成功的元素，然后把数据填充到req对象的字段里。

从例子上看会更清晰一些：

```
app.get('/user/:id', function(req, res){
  res.send('user ' + req.params.id);
});
```

/user/:id这样的URL有一个占位符叫做id。Express会识别/user/后的内容，然后将其注册到req.params.id字段上。如果你喜欢，匹配方式可以用正则表达式。

2. 数学计算服务器（和客户端）

现在，我们创建一个支持数学运算的服务器。服务器的返回结果用JSON对象表示。它一样支持前面Math Wizard里的4种操作。

创建一个math-server.js，使其包含下面的代码：

```
var math  = require('./math');
var express = require('express');
var app = express.createServer(
  //express.logger()
);
app.configure(function(){
  app.use(app.router);
  app.use(express.errorHandler({
    dumpExceptions: true, showStack: true }));
});
```

```
app.get('/fibonacci/:n', function(req, res, next) {
  math.fibonacciAsync(Math.floor(req.params.n),
  function(val) {
    res.send({ n: req.params.n, result: val });
  });
});
app.get('/factorial/:n', function(req, res, next) {
  res.send({
    n: req.params.n,
    result: math.factorial(Math.floor(req.params.n))
  });
});
app.get('/mult/:a/:b', function(req, res, next) {
  res.send({
    a: req.params.a, b: req.params.b,
    result: req.params.a * req.params.b
  });
});
app.get('/square/:a', function(req, res, next) {
  res.send({
    a: req.params.a,
    result: req.params.a * req.params.a
  });
});
app.listen(3002);
```

上面部分就是除了数学模块之外的整个服务器，和之前我们使用的一样。它有一个比较简单的配置，通过监听http://localhost:3002/使其作为Math Wizard的后台程序。

我们指定的路由是很简单的，作用仅仅是为每一个操作需要的函数参数提供存放空间。

这是我们第一次看到`res.send`的使用。直接以数组形式的HTTP报头信息值（特定于HTTP响应报头）和HTTP状态码来发送响应数据是一个很灵活的方式。它在这里自动检测对象，将其格式化为JSON格式的文本，然后用正确的Content-Type发送。

4

现在让我们运行一下看看：

```
$ node math-server.js &
[1] 10483
$ curl -f http://localhost:3002/square/34.2
{"a":"34.2","result":1169.64}
$ curl -f http://localhost:3002/mult/3.3/3
{"a":"3.3","b":"3","result":9.899999999999999}
$ curl -f http://localhost:3002/factorial/20
{"n":"20","result":2432902008176640000}
$ curl -f http://localhost:3002/fibonacci/20
{"n":"20","result":6765}
```

现在我们已经实现了服务器，那么客户端要怎么做呢？

因为这次我们要开发的是HTTP服务，所以客户端程序需要创建到服务器的HTTP请求。Node包含了一个非常好的HTTP客户端对象，在第5章我们会进行深入探讨。

我们的任务是创建一个HTTP请求，发送请求，然后等待响应，将响应数据进行解码，然后

使用它。既然可以在类似Math Wizard这样的Web应用里实现这一任务，我们可以创建一个简单的、对应数学服务器的客户端应用。

创建一个包含下面代码的math.client.js文件：

```
var http = require('http');
var util = require('util');
[
  "/fibonacci/20", "/factorial/20",
  "/mult/10/20", "/square/12"
].forEach(function(path) {
  var req = http.request({
    host: "localhost",
    port: 3002,
    path: path,
    method: 'GET'
  }, function(res) {
    res.on('data', function (chunk) {
      util.log('BODY: ' + chunk);
    });
  });
  req.end();
});
```

http.request方法会创建一个HTTP请求，并将URL元素分割到参数对象中。对此，在下一章的内容里，我们会有更深入的探讨，不过现在你需要知道res.on语句里声明的回调函数会在HTTP响应数据到达时被触发。

之后math-client.js会创建一些对应math-server.js的硬编码请求，输出如下的内容：

```
$ node math-client.js
7 Jun 22:17:49 - BODY: {"n":"20","result":2432902008176640000}
7 Jun 22:17:49 - BODY: {"a":"12","result":144}
7 Jun 22:17:49 - BODY: {"a":"10","b":"20","result":200}
7 Jun 22:17:49 - BODY: {"n":"20","result":6765}
```

眼尖的读者可能会注意到响应数据到达的顺序和它们在数组里的顺序并不一致。虽然斐波那契计算请求最先发送，但是对应的响应却最后达到。回忆下，回调函数的异步执行以它们到达事件循环的时间为基础，而斐波那契函数就有一些时间计算结果。事实上，斐波那契函数在计算第20个数字的序列时会消耗很长的时间。在math-client.js里所有的请求都会很快地被发送出去，因为它本身发送请求的工作量是很小的，而我们看到的输出结果来自处理函数res.on('data'..)。math.server.js里的请求是交付给app.get请求处理函数的。每一个res.on('data'..)处理函数都会通过套接字（负责发送HTTP请求）和一个app.get请求处理函数绑定。每当app.get请求处理函数调用res.send，它的HTTP响应会反过来让res.on('data'..)处理函数等待响应的产生。

那么，是什么决定了输出的顺序呢？其实是math-server.js用来计算每个结果所消耗的时间，因为结果只会在响应到达时才被输出。

大部分情况下，计算（如乘法）的速度是很快的，服务器能很快返回计算结果。我们之前讨

论的斐波那契查询里的问题又是另外一回事了。因为fibonacciAsync的使用，斐波那契值的计算会与运行其他计算并行进行。而第20个斐波那契数的计算会消耗很长时间，因此其他先计算出的值会首先到达客户端。将斐波那契请求的值改为2会缩短计算时间，从而改变响应到达的顺序，就像下面这样：

```
$ node math-client.js
7 Jun 22:34:49 - BODY: {"n":"2","result":1}
7 Jun 22:34:49 - BODY: {"a":"10","b":"20","result":200}
7 Jun 22:34:49 - BODY: {"n":"20","result":2432902008176640000}
7 Jun 22:34:49 - BODY: {"a":"12","result":144}
```

3. 使用数学服务器重构Math Wizard应用

现在我们已经有了一个客户端函数，将它移植到Math Wizard里的请求处理函数中是相当容易的。之前我们考虑了直接面对用户的服务器如何做到用户体验更好的请求处理，同时又能处理一个庞大的计算。斐波那契序列的计算就是一个大型计算的例子，如果放到用户直接面对的服务器上计算，肯定会降低用户使用时的愉悦度。

我们之前的解决方案是通过将计算切分成多块的方式重构算法。虽然这个方法在一些情况下有效，但是面向用户的服务器依然需要进行计算，而重构的算法可能更低效。在使用math-client.js时，我们有另一种方式来解决这个问题，即将计算工作发送到一个后台服务器或者使用负载均衡的服务器集群。

在app-express.js里，我们用下面的代码替代/fibonacci请求处理函数：

```
app.get('/fibonacci', htutil.loadParams, function(req, res) {
  if (req.a) {
    var httpreq = require('http').request({
      host: "localhost",
      port: 3002,
      path: "/fibonacci/"+Math.floor(req.a),
      method: 'GET'
    }, function(httpresp) {
      httpresp.on('data', function (chunk) {
        var data = JSON.parse(chunk);
        req.result = data.result;
        res.render('fibo.html',
          { title: "Math Wizard", req: req });
      });
    });
      httpreq.end();
      //math.fibonacciAsync(Math.floor(req.a), function(val) {
        //req.result = val;
        //res.render('fibo.html',
          //{ title: "Math Wizard" , req: req });
      //});
  } else {
    res.render('fibo.html',
      { title: "Math Wizard" , req: req });
  }
});
```

4

新的请求处理函数会回过头来从我们刚刚实现的后台服务器（`math-server.js`）创建一个HTTP请求。实际上这是你可以想象的最简单的REST式后台服务。后台服务器为HTTP GET请求定义了一些URL，它会返回JSON格式的数据，然后我们的请求处理函数会解析JSON数据来获取结果。

在修改请求处理函数后，你还可以用一样的方式运行：

```
$ node app-express.js
```

这就好像在使用`fibonacciAsync`，结果没有变，那么区别在哪里呢？为什么要用后台服务呢？在Math Wizard里，这可能有些过分使用了，但是它证明了一个对你的应用来说可能比较完美的选择。下面给出一些理由。

- 最好把繁重的计算需求从直接面对用户的服务器移除，把它的资源留给与浏览器交互。
- 用负载均衡做到用多个服务器处理请求（云计算）。
- `math-server.js`的响应能让一个缓存代理的使用变成一个非常吸引人的、急剧提升速度的途径。毕竟，为什么要重新运行已有结果的计算呢？
- 它可以让你通过慷慨陈词打动你的老板。

这个方式让我们更容易地实现`math-server.js`并将其整合到Math Wizard里，它证明了Node在各方面带来的简化能力和强大功能。

4.4 小结

我们在这一章学习了很多内容，现在可以准备做一些更贴近实际的应用了。下面先回顾一下本章的知识：

- 处理请求和模块化管理；
- 使用HTTP Server 对象，并结合Connect和Express创建web应用；
- 处理表单提交的URL查询参数；
- 运行时间很长的计算给服务器响应和用户满意度带来的影响，以及对应的解决方法；
- 使用异步模块编写异步代码；
- Connect和Express为一个完整Web应用提供的各种支持；
- Connect作为中间件的价值所在；
- 如何用Connect和Express路由规则处理不同的HTTP动作；
- 在Express里使用参数化的URL；
- 实现一个REST式后台服务器完成负载均衡。

到现在为止，我们已经学习了很多与实现Web应用相关的内容，下一章将对HTTP服务器、客户端对象，以及Node系统里事件的分配进行更加深入的探讨。

第 5 章

简单的Web服务器、EventEmitter和HTTP客户端

我们已经知道了如何使用Express框架创建Node应用程序，接下来将深入探索HTTP Web服务器的实现细节。在这一章，我们将实现一个简易的Web服务器，它具备真实Web服务器的功能，这些功能在第4章讨论过。

HTTP协议实现起来相当复杂，这些细节最好还是交给Web应用框架去做。既然如此，为何还要实现自己的HTTP Web服务器呢？下面是几点理由：

- 理解如何选用框架；
- 理解框架的设计；
- 框架不能满足所有任务的需要；
- 需要在HTTP层直接编码实现Web服务，而不是Web应用；
- 你的想法比框架作者更高明。

开始吧。

5.1 通过 EventEmitter 发送和接收事件

在Node应用中，`EventEmitter`对象是非常关键的部分。它是如此基础，以至于你可能还未察觉到它的存在。Node中的很多类都是`EventEmitter`的子类，使用`EventEmitter`的方法发送事件来表示一些状态，这些事件通过Node 的事件循环，最终触发回调函数。

在这一章中，我们将使用`HTTPServer`和`HTTPClient`类，它们都继承自`EventEmitter`，因此在HTTP协议的每一步，它们都基于EventEmitter发送事件。理解EventEmitter能帮助我们理解这两个对象，以及Node中的许多其他对象。

`EventEmitter`被定义在Node的事件（`events`）模块中，直接使用`EventEmitter`类意味着需要先声明`require('events')`，除非使用场景确实需要这么做，否则不必显式声明`require('events')`，因为Node中很多对象都无需你调用`require('events')`就会使用EventEmitter.

下面的例子（`pulser.js`）直接使用了`EventEmitter`类，它会告诉我们如何发送和接收事件。

```
var events = require('events');
var util   = require('util');

function Pulser() {
  events.EventEmitter.call(this);
}
util.inherits(Pulser, events.EventEmitter);

Pulser.prototype.start = function() {
  var self = this;
  this.id = setInterval(function() {
    util.log('>>>> pulse');
    self.emit('pulse');
    util.log('<<<< pulse');
  }, 1000);
}
```

例子定义了一个类Pulser，该类（通过util. inherits）继承自EventEmitter。它的作用是每隔一秒钟向所有监听器发送一个定时事件。start方法使用了setInterval这个函数来定期（每秒钟一次）重复执行回调函数，并调用emit方法将pulse事件发送给每一个监听器。

Pulser.js的大部分代码都可以作为一个独立模块，供任何需要提供定时服务的应用使用。

接下来看看如何使用Pulser对象：

```
var pulser = new Pulser();
pulser.on('pulse', function() {
  util.log('pulse received');
});
pulser.start();
```

在这里我们创建了一个Pulser对象并处理其pulse事件。执行pulser.on('pulse'..)为pulse事件和回调函数建立联系。然后这段代码执行start方法让整个过程跑起来。

把这些代码敲完后保存为pulser.js，然后运行，你就可以看到如下的输出结果：

```
$ node pulser.js
23 May 20:30:20 - >>>> pulse
23 May 20:30:20 - pulse received
23 May 20:30:20 - <<<< pulse
23 May 20:30:21 - >>>> pulse
23 May 20:30:21 - pulse received
23 May 20:30:21 - <<<< pulse
...
```

EventEmitter 原理

EventEmitter发出的事件都有名称，比如例子中用到的pulse。事件名称可以是任何有意义的词，而你可以根据需要定义多个事件。事件名称可以通过简单地调用.emit方法并提供事件名来设置。这里不需要注册事件名那样正式的操作，调用.emit方法就够了。按照惯例，事件名error用于表示发生错误时对应的事件。

对象使用.emit函数发送事件。所有注册到对应事件的监听器都可以收到事件。我们可以通

过调用.on方法注册监听器，参数是事件名，并用一个回调函数接收事件。

在pulse.js中我们可以看到这一点。Pulser对象调用self.emit('pulse')来发送事件，稍后pulse.on('pulse',..)被执行并用于接收事件。

通常来说，有一些数据需要伴随着事件同时发送。为此，我们只需像这样把要传递的数据用作.emit执行时的参数：

```
self.emit('eventName', data1, data2, ..);
```

程序会接收到这个事件，而被传递的数据会作为回调函数的参数传入，下面的代码可以监听这种事件：

```
emitter.on('eventName', function(data1, data2, ..) {
  // 接收到事件后的操作
});
```

从下一节起，我们会看到一些实际应用时的例子（使用HTTP对象），HTTP客户端对象和服务器对象都是EventEmitter，它们在HTTP协议的各个阶段发送相应的事件。例如，每个到来的HTTP请求都被封装成一个HTTP Request对象，当有请求数据到达时，这个对象发送data事件，并在所有数据都到达时发送end事件，假如发送end事件之前套接字连接关闭，它将发送close事件。

5.2 HTTP Sniffer——监听 HTTP 会话

现在让我们使用HTTP对象创建一个实用的类，用这个类监听HTTP服务器对象发出的所有事件。该类可以做一个实用的调试工具，也演示了HTTP服务器对象的工作原理。

Node中的HTTP服务器对象是一个EventEmitter，HTTP Sniffer所做的只是监听每个服务器事件，然后输出每个事件的相关信息。

创建一个名为httpsniffer.js的文件，为其填入以下内容：

```
var util = require('util');
var url  = require('url');

exports.sniffOn = function(server) {
  server.on('request', function(req, res) {
    util.log('e_request');
    util.log(reqToString(req));
  });

  server.on('close', function(errno) {
    util.log('e_close errno='+ errno);
  });

  server.on('checkContinue', function(req, res) {
    util.log('e_checkContinue');
    util.log(reqToString(req));
    res.writeContinue();
  });
```

5

```
    server.on('upgrade', function(req, socket, head) {
      util.log('e_upgrade');
      util.log(reqToString(req));
    });

    server.on('clientError', function() {
      util.log('e_clientError');
    };

    // server.on('connection', ..);
  }

  var reqToString = function(req) {
    var ret = 'request ' + req.method +' '+
      req.httpVersion +' '+ req.url +'\n';
    ret += JSON.stringify(url.parse(req.url, true)) +'\n';
    var keys = Object.keys(req.headers);
    for (var i = 0, l = keys.length; i < l; i++) {
      var key = keys[i];
      ret += i +' '+ key +': '+ req.headers[key] +'\n';
    }
    if (req.trailers)
      ret += req.trailers +'\n';
      return ret;
  }
  exports.reqToString = reqToString;
```

上面的代码比较多，关键部分是`sniffOn`函数。给定一个HTTP服务器函数时，它使用`.on`方法连接监听函数，后者输出HTTP服务器对象发送的每个事件的相关数据。此对象的事件对应HTTP协议中服务器与客户端之间的数据交换。

下面是使用HTTP Sniffer的一个例子，它由hello world那个例子（`hwserver.js`）修改而来：

```
var http    = require('http');
var sniffer = require('./httpsniffer');

var server = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
});
sniffer.sniffOn(server);
server.listen(3000);
```

有了这些之后，运行该服务器：

```
$ node hwserver.js
```

在浏览器中访问http://localhost:3000/，你就能看到如下的控制台输出。注意有两个请求产生了，一个对应`/`，另一个对应`/favicon.`。`favicon`就是一个显示在浏览器中方便标识你的网站的小图标。此刻我们使用的服务器还不支持`favicon`，稍后我们会了解如何实现它。

```
$ node hwserver.js
6 Apr 21:14:38 - e_request
```

```
6 Apr 21:14:38 - request GET 1.1 /
{"search":"","query":{},"pathname":"/","href":"/"}
0 host: localhost:3000
1 user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us)
AppleWebKit/533.20.25 (KHTML, like Gecko) Version/5.0.4 Safari/533.20.27
2 accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,
image/png,*/*;q=0.5
3 cache-control: max-age=0
4 accept-language: en-us
5 accept-encoding: gzip, deflate
6 connection: keep-alive

6 Apr 21:14:39 - e_request
6 Apr 21:14:39 - request GET 1.1 /favicon.ico
{"search":"","query":{},"pathname":"/favicon.ico","href":"/favicon.ico"}
0 host: localhost:3000
1 user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us)
AppleWebKit/533.20.25 (KHTML, like Gecko) Version/5.0.4 Safari/533.20.27
2 referer: http://localhost:3000/
3 cache-control: max-age=0
4 accept: */*
5 accept-language: en-us
6 accept-encoding: gzip, deflate
7 connection: keep-alive
```

现在有这个工具可以嗅探HTTP服务器事件了。这个简易的技术能输出事件的详细日志，这样的模式适用于所有EventEmitter对象。在程序中，你可以使用该技术检查EventEmitter对象的真实行为。

5.3　基本的 Web 服务器

本节介绍了一个基本Web服务器Basic Server的实现。Node提供了一个很好的HTTP服务器对象，若再结合使用一些额外的协议元素和服务，即可提供网站的基本特性。

Basic Server非常基本，它实现了以下特性：

- 灵活的请求路由选择；
- 自动提供解析后的URL对象；
- 自动提取主机头信息（用于虚拟主机）；
- 自动提取cookie头信息；
- 支持favicon.ico请求；
- 支持静态文件（HTML、JS、PNG、GIF、JPEG等）；
- 灵活的服务器配置。

有了这些作为目标，Basic Server的实现代码由4个Node模块、一个CSS文件、一个或数个HTML文件组成。如此小巧的尺寸也印证了Node的灵活和强大。

预先有个准备工作，那就是安装MIME模块，用于生成正确的Content-Type头。可能你对MIME

感兴趣，我们稍后会详细讨论。现在，敲入这个命令：

```
$ npm install mime
```

Basic Server 的实现

在写代码之前，让我们先思考一下实现上述目标的策略。Node创造性地提供了这样的服务器架构：

```
var server = http.createServer(function (req, res) {
  // 处理请求
});
server.listen(port);
```

我们的目标是实现一个HTTP请求处理函数，它会检查每个请求，然后基于请求属性，给予每个请求适当的响应。这个设计架构可以将请求的检查和派发从应用程序的业务逻辑中分离出来。

1. Basic Server核心（**basicserver.js**）

Basic Server的核心模块会创建一个HTTP服务器对象，附加Basic Server上用于检查请求，然后给予适当响应的功能。

创建名为`basicserver.js`的文件，为其键入以下内容：

```
var http    = require('http');
var url     = require('url');

exports.createServer = function() {
  var htserver = http.createServer(function(req, res) {
    req.basicServer = {
      urlparsed: url.parse(req.url, true)
    };
    processHeaders(req, res);
    dispatchToContainer(htserver, req, res);
  });
  htserver.basicServer = { containers: [] };
  htserver.addContainer = function(host, path,
                                   module, options) {
    if (lookupContainer(
      htserver, host, path) !== undefined) {
      throw new Error("Already mapped "+host+"/"+path);
    }
    htserver.basicServer.containers.push({
      host: host, path: path,
      module: module, options: options });
    return this;
  }
  htserver.useFavIcon = function(host, path) {
    return this.addContainer(host, "/favicon.ico",
      require('./faviconHandler'),
      { iconPath: path });
  }
  htserver.docroot = function(host, path, rootPath) {
```

```
    return this.addContainer(host, path,
      require('./staticHandler'),
      { docroot: rootPath });
  }
  return htserver;
}
```

在Basic Server的这些核心代码中，有一个createServer函数。我们在给服务器增加功能时用createServer创建并返回一个HTTP服务器对象。这里主要展示的是请求处理函数。我们的策略是先在请求对象中添加有用的信息（在processHeaders函数中处理），然后将其附加到合适的处理函数中（在dispatchToContainer函数中处理）。第二个模块也将被使用同样的策略配置你的服务器。稍后，我们将看到一个这样的服务器配置模块。

我们的一个策略是将有用的数据同时添加到服务器（htserver）和请求（req）对象中。因为JavaScript是松散类型的语言，我们可以完成添加的步骤。所有的附加操作都在basicServer对象中进行，并在这个函数中附加数据到htserver和req中。通过这样的方式我们可以添加任何数据到htserver上，因为数据隐藏在htserver.basicServer对象中，所以其他代码无法访问到这些数据。

我们做的另外一件事是添加3个函数来管理一系列容器。容器大致对应我们在上一章提到的Express路由器中间件。这3个函数添加一个容器到服务器（addContainer），然后配置两个内置的容器，一个处理favicon（useFavIcon），另一个处理静态文件（docroot）。

容器由下面4块数据定义：

- 一个用于匹配Host头的正则表达式；
- 一个用于匹配请求URL的正则表达式；
- 一个选项对象；
- 一个处理函数。

综合上面的策略就可以实现一个基于名字的虚拟主机，意味着Basic Server可以通过判断Host头部匹配的容器对象响应来自多个域名的请求。之后我们会讨论更多相关的内容。

options对象用于帮助将配置模块的配置数据传递到处理函数模块，而options对象的内容是由处理函数模块定义的。

比如在favicon处理函数中，options对象包含favicon图片的路径。浏览器请求favicon时路径一般都是/favicon.ico结尾，这个路径是硬编码在容器里的。

在前面的例子里我们还使用了其他一些函数，但是还没有对其进行说明。第一个是lookupContainer，用于检查containers数组中是否有容器匹配HTTP请求中的host和path字段。

```
var lookupContainer = function(htserver, host, path) {
  for (var i = 0;
    i < htserver.basicServer.containers.length; i++) {
    var container = htserver.basicServer.containers[i];
    var hostMatches = host.toLowerCase().match(container.host);
    var pathMatches = path.match(container.path);
    if (hostMatches !== null && pathMatches !== null) {
      return {
```

```
        container: container,
          host: hostMatches,
          path: pathMatches };
      }
    }
    return undefined;
  }
```

这个函数通过正则表达式匹配数组中的host和path，实现了一个简单的扫描。如果找到匹配项则返回该项，否则直接返回undefined。

下一个函数processHeaders用于搜索req.headers数组以查找cookie和Host头部，因为这两个字段对请求的分派都很重要。和你在前面看到的请求处理函数部分一样，这一函数在每一个HTTP请求到达时都会被调用。

```
var processHeaders = function(req, res) {
    req.basicServer.cookies = [];
    var keys = Object.keys(req.headers);
    for (var i = 0, l = keys.length; i < l; i++) {
        var hname = keys[i];
        var hval  = req.headers[hname];
        if (hname.toLowerCase() === "host") {
            req.basicServer.host = hval;
        }
        if (hname.toLowerCase() === "cookie") {
            req.basicServer.cookies.push(hval);
        }
    }
}
```

还有很多其他的HTTP头部字段（Accept、Accept-Encoding、Accept-Language和User-Agent）可能需保存以供后续使用，具体视你的应用而定。

最后一个函数是dispatchToContainer，它的功能就是查找匹配的容器，分派请求到对应的容器中。和processHeaders一样，这个函数会在每一个HTTP请求到达时被调用。

```
var dispatchToContainer = function(htserver, req, res) {
  var container = lookupContainer(htserver,
    req.basicServer.host,
    req.basicServer.urlparsed.pathname);
  if (container !== undefined) {
    req.basicServer.hostMatches = container.host;
    req.basicServer.pathMatches = container.path;
    req.basicServer.container   = container.container;
    container.container.module.handle(req, res);
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("no handler found for "+
      req.host +"/"+ req.urlparsed.path);
  }
}
```

如果没有找到对应的容器，用户就会得到一个错误页面（错误状态码404）。

处理函数模块暴露了函数handle，参数是req和res。在dispatchToContainer中，Basic

Server通过调用handle派发请求。

2. favicon处理函数（faviconHandler.js）

Basic Server包含两个我们还没有提到的内建处理函数模块，第一个是favicon处理函数faviconHandler.js，用来响应favicon请求。当配置模块使用了useFavIcon函数，这个模块就会被安装到Basic Server上：

```
var fs = require('fs');
var mime = require('mime');
exports.handle = function(req, res) {
  if (req.method !== "GET") {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("invalid method " + req.method);
 } else if (req.basicServer.container.options.iconPath!== undefined) {
    fs.readFile(req.basicServer.container.options.iconPath,
    function(err, buf) {
      if (err) {
        res.writeHead(500, {
          'Content-Type': 'text/plain' });
        res.end(
          req.basicServer.container.options.iconPath
          +" not found");
      } else {
        res.writeHead(200, {
          'Content-Type':
          mime.lookup(req.basicServer.container.options.iconPath),
          'Content-Length': buf.length
        });
        res.end(buf);
      }
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("no favicon");
  }
}
```

这个处理函数处理对favicon.ico的请求。

重申一遍，处理函数模块需要在export下注册一个名为handle的函数function(req, res)。Basic Server找出匹配请求的容器，然后执行处理函数的handle函数。这个处理函数读取iconPath指定的文件，然后使用res对象将其发送到浏览器。检测多个错误条件后，处理函数使用res发送错误页面。

MIME模块根据给出的图标文件确定正确的MIME类型。网站图标favicon可以是任何类型的图片，但是我们必须要告诉浏览器是哪个类型。

除了GET请求，该处理函数对其他请求都无效。所以它检查网络请求的方法，对GET请求之外的请求返回一个状态码为404的响应。

3. 静态文件处理程序（staticHandler.js）

现在看一下必需的响应代码（响应对.html或.css等文件的请求）。创建一个名为static-

Handler.js的文件，为其键入以下内容：

```
var fs   = require('fs');
var mime = require('mime');
var sys  = require('sys');
exports.handle = function(req, res) {
  if (req.method !== "GET") {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end("invalid method " + req.method);
  } else {
    var fname = req.container.options.docroot +
      req.urlparsed.pathname;
    if (fname.match(/\/$/)) fname += "index.html";
    fs.stat(fname, function(err, stats) {
      if (err) {
        res.writeHead(500, {
          'Content-Type': 'text/plain' });
        res.end("file "+ fname +" not found " + err);
      } else {
        fs.readFile(fname, function(err, buf) {
          if (err) {
            res.writeHead(500, {
              'Content-Type': 'text/plain' });
            res.end("file "+
              fname +" not readable " + err);
          } else {
            res.writeHead(200, {
              'Content-Type':
              mime.lookup(fname),
              'Content-Length': buf.length
              });
            res.end(buf);
          }
        });
      }
    });
  }
}
```

这个例子展示了一个静态文件处理程序static Handler，用于处理文件系统内的文件。

docroot选项指被存放文件所在文件夹的路径。staticHandler读取docroot目录下的指定文件，如果文件存在并且读取时未发生错误，文件内容会被包含在res对象中发送到浏览器，这一任务非常简单。

一个特殊情况是使用MIME模块（可以通过npm获得该模块）确定正确的Content-Type头。MIME类型是必需的，以便浏览器正确解析数据，稍后我们会进一步谈论这个问题。

另一种特殊情况是，如果请求URL以/结尾，那么处理程序在请求地址上追加index.html。

4. Basic Server的配置文件 (server.js)

我们完成了Basic Server的所有组件，现在可以打造一个可以工作的Web服务器了。创建一个名为server.js的文件并为其键入以下内容：

```
var port = 4080;
var server = require('./basicserver').createServer();
server.useFavIcon("localhost", "./docroot/favicon.png");
server.docroot("localhost", "/", "./docroot");
require('./httpsniffer').sniffOn(server);
server.listen(port);
```

这一配置文件中指定了一个名为`docroot`的文件夹作为静态文件的根目录，这个目录下，名为`favicon.png`的图片被指定为网站图标。换句话说，我们已经配置了一个简易的无动态页面的Web服务器。

HTTP Sniffer已经连接上，因此对于浏览器发出的每个请求，其详细信息都会出现在控制台。

在运行服务器之前，让我们先看看要放在`docroot`目录下的内容。

增加几个HTML文件有利于测试服务器，这些HTML文件的内容可以像这样编写（可以命名为`index.html`）：

```
<html>
<head>
  <link href="/style.css" rel="stylesheet">
</head>
<body>
  <h1>Index</h1>
  <p><a href="page2.html">page 2</a></p>
  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam
    fringilla molestie leo eu tincidunt. Donec pulvinar porttitor
    dictum. Fusce at elit mauris, a ornare ipsum. Nulla congue nisi
    non ante pellentesque vel lobortis lacus varius. Nam metus ante,
    blandit in rutrum et, pellentesque eu velit. Nulla blandit
    placerat scelerisque. Morbi odio magna, accumsan sit amet
    pharetra eu, varius sit amet ipsum. Aenean interdum libero ut est
    hendrerit dictum. Suspendisse convallis pellentesque metus
    ac tempor. Nam   diam lectus, posuere eu rutrum id, facilisis vel
    tellus.
  </p>
</body>
```

你可以使用喜欢的文本自动生成工具（http://www.lipsum.com/）创建多个类似的HTML文件。为了方便，多个HTML文件之间可以通过`<a>`标签建立链接。

该HTML文件引入了一个名为`style.css`的CSS文件，CSS内容如下：

```
body {
  color: #00c;
  font-family: Verdana, Arial, Helvetica, sans-serif;
  background-color: #cf9
}
H1 {
  color: #ff6;
  background-color: #090;
  border: solid 5px #0f9
}
```

最后，我们创建一个名为`favicon.png`的小图片文件。网站图标favicon是浏览器显示在地

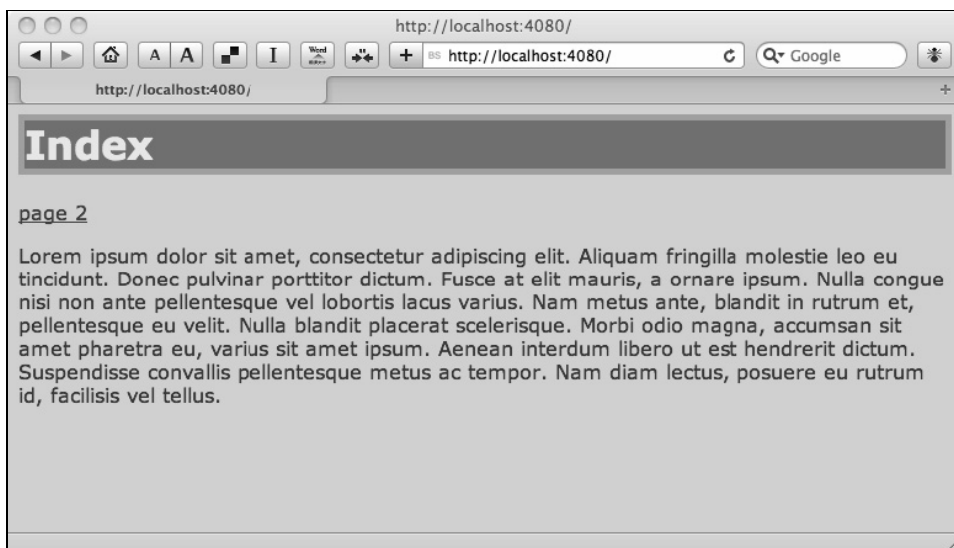
址栏上的小图片，根据维基百科的解释（http://en.wikipedia.org/wiki/Favicon），它是32 × 32或者48 × 48大小（单位为像素）的几乎任意格式的图片，能正常显示在除了IE浏览器（它坚持使用ICO文件）之外的所有浏览器上。

现在启动Basic Server:

```
$ node server.js
```

在Web浏览器中访问http://localhost:4080。

恭喜！你已经启动了Basic Server。如图所示，浏览器会显示`docroot`目录下`index.html`的内容。



Basic Server极易扩展，它可以：

- 设置多个虚拟域；
- 添加自定义处理程序；
- 支持cookie头；
- 实现HTTP基本验证[①]和支持HTTPS请求。

5. Basic Server的虚拟主机配置

使用虚拟主机是一个常见的需求。假如需要支持额外的域名，你可能会像下面这样为每个虚拟域配置一个文件夹：

```
// 两个域名独立，内容也独立
bs.useFavIcon("example.com", "./example.com/favicon.png");
bs.docroot("example.com", "/", "./example.com");
```

① 在HTTP中，基本验证用来允许Web浏览器或其他客户端程序在请求时提供用户名和口令形式的凭证，详见http://zh.wikipedia.org/zh-sg/HTTP%E5%9F%BA%E6%9C%AC%E8%AE%A4%E8%AF%81。

```
bs.useFavIcon("example2.com", "./example2.com/favicon.png");
bs.docroot("example2.com", "/", "./example2.com");
// 将一个域名挂载在另一个上
bs.useFavIcon("parked.com", "./example.com/favicon.png");
bs.docroot("parked.com", "/", "./example.com");
```

如例所示，我们可以将一个域挂载在另一个域上（配置两个域使其访问同一个容器），也可以如下使用正则表达式来配置：

```
bs.useFavIcon("parked.com|example.com",
  "./example.com/favicon.png");
bs.docroot("parked.com|example.com", "/", "./example.com");
```

6. 一个用于Basic Server的shorturl模块

常见的需求不是将一个域挂载在另一个域上，而是把对一个域的请求重定向到另一个上。例如，将www.example.com重定向到example.com（移除www）上。另一个例子是提供类似tinyurl.com的服务，使用简短的URL跳转到较长的URL.。

实现这两种情况，我们需要在HTTP响应中发送301（永久移除）或者302（临时移除）状态码，并且指定Location头信息。有了这个组合信号，Web浏览器就知道要跳转到另一个Web位置了。

我们来为Basic Server实现一个短域名处理模块，它能为一份代码执行302跳转。创建一个名为`redirector.js`的文件：

```
var util = require('util');
var code2url = {
  'ex1': 'http://example1.com',
  'ex2': 'http://example2.com',
};
var notFound = function(req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end("no matching redirect code found for "+
    req.basicServer.host +"/"+
    req.basicServer.urlparsed.pathname);
}
exports.handle = function(req, res) {
  if (req.basicServer.pathMatches[1]) {
    var code = req.basicServer.pathMatches[1];
    if (code2url[code]) {
      var url  = code2url[code];
      res.writeHead(302, { 'Location': url });
      res.end();
    } else {
      notFound(req, res);
    }
  } else {
    notFound(req, res);
  }
}
```

这是一个可以在Basic Server中使用的处理程序模块。我们需要在`server.js`中加入一行配置（配置`docroot`容器之前）：

```
server.addContainer(".*", "/l/(.*)$",  require('./redirector'), { });
```

我们在配置容器的主机和路径两个部分都用了正则表达式。正则表达式.*可以匹配任何主机名，匹配路径名的正则表达式能匹配以/1/开头的路径，并将路径的其余部分记为子匹配。

当访问http://localhost:4080/l/code1时，路径名称匹配数据会出现在req.basicServer.pathMatches中，而/l/后面的部分则被存入req.basicServer.pathMatches[1]中。如果所有部分匹配成功，处理程序模块会返回一个状态码为302的HTTP响应，响应的位置头信息会包含从code2url对象检索到的URL。

5.4　MIME 类型和 MIME npm 包

根据HTTP协议实现一个成功且正确的Web服务器，需要处理很多细节，其中一个细节就是Content-Type头，它借用自MIME协议。

MIME协议设计于20世纪90年代初，原本用于改进电子邮件的功能。HTTP协议与其在同一时间段被设计出来，它们面临相同的挑战，那就是标识电子邮件附件或HTTP请求的数据格式。文件扩展名不足以完全恰当地标识文件类型，因为3个（左右的）字符作为标识符来说太短，而且文件扩展名没有标准。于是，人们设计了Content-Type头和整个MIME类型标准来作为数据类型的表示系统，并保证MIME类型同时适用于电子邮件和HTTP。

抛开历史原因，Content-Type头是必需的。问题是应用程序如何知道该发送哪一种Content-Type。对于某些应用，特别是一些处理固定数据的小型应用，我们可以精准地知道该使用哪一种Content-Type头，因为应用发送的数据是特定已知的。

然而staticHandler能发送任何文件，且通常不知道该使用哪种Content-Type。通过匹配文件扩展名列表和Content-Type可以解决这个问题，但跟早先说过的一样，这个方案并不完美。最好的实践方案是使用一个外部的配置文件，它通常由操作系统提供。

MIME npm包使用了Apache项目的mime.types文件，该文件包含超过600个Content-Type的有关数据。如果有需要，mime模块也支持添加自定义的MIME类型。

安装模块：

```
$ npm install mime
```

一段测试代码：

```
var mime = require('mime');
var mimeType = mime.lookup('image.gif'); // ==> image/gif
res.setHeader('Content-Type', mimeType);
```

有些相关的HTTP头 (参见http://www.w3.org/Protocols/ 和 http://en.wikipedia.org/wiki/List_of_HTTP_header_fields)你可能喜欢：

- Content-Encoding　数据被编码时使用，例如gzip；
- Content-Language　内容中使用的语言；
- Content-Length　字节数；
- Content-Location　能取到数据的一个候补位置；
- Content-MD5　内容主体的MD5校验和。

5.5　处理 cookie

另一个重要的特性是支持cookie。HTTP协议是无状态的，这意味着Web服务器不能辨认不同的请求发送端。那么在协议不支持状态这个概念时，我们如何登录一个网站呢？普遍的做法是，服务器发送cookie到客户端浏览器，cookie中定义了登录用户的身份。对于每一次请求，Web浏览器都会发送对应所访问网站的cookie。

Basic Server对识别浏览器发出的cookie做到了部分支持，请求处理程序通览req对象的头信息，识别cookie头，然后将其保存到一个数组中。（参见http://en.wikipedia.org/wiki/HTTP_Cookie）。

下面的代码能提取浏览器发送的cookie，并解析hval变量从字符串中提取cookie值：

```
var keys = Object.keys(req.headers);
for (var i = 0, l = keys.length; i < l; i++) {
  var hname = keys[i];
  var hval = req.headers[hname];
  if (hname.toLowerCase() === "cookie") {
    req.basicServer.cookies.push(hval);
  }
}
```

发送cookie时，我们应以如下方式为Set-Cookie或Set-Cookie2头设一个值：

```
res.setHeader('Set-Cookie2', .. cookie value ..);
```

cookie是结构化的文本格式，在Basic Server（或Connect）这样的Web框架中，cookie的字符串解析和格式化是一个备选功能。以下是现有的几个相关库：

- https://github.com/jed/cookies/——提供近乎完整的cookie处理和验证，支持签名cookie；
- https://github.com/bmeck/node-cookiejar——一个小巧的cookie解析库。

5.6　虚拟主机和请求路由

建立虚拟主机是在同一IP地址上托管多个域名的方法。如Basic Server所示，Node可以实现基于域名的虚拟主机。

对于基于域名的虚拟主机，HTTP请求会包含一个域名头信息：

```
GET /path/to/request HTTP/1.1
Host: example.com
```

Node中的req对象包含一个名为headers的数组，该数组中包含了主机头。如Basic Server所示，虚拟主机易于实现，只需检查请求中的headers数组，然后映射请求到适当的域。

5.7　发送 HTTP 客户端请求

我们已经深入研究过HTTP服务器对象，现在我们来看它的客户端对象。Node包含一个HTTP客户端对象，它用来发送HTTP请求，而且是任何类型的HTTP请求。但是它不能仿效一个全功能

的浏览器，所以不要幻想它是一个健壮的测试自动化工具。使用它可以打造浏览器模拟器或者其他一些HTTP客户端。例如，任何REST式Web服务都能通过HTTP客户端对象调用。

受到`wget`和`curl`命令的启发，接下来写一段可以发送HTTP请求并显示结果的代码。创建一个名为`wget.js`的文件，为其写入如下代码：

```
var http = require('http');
var url  = require('url');
var util = require('util');

var argUrl = process.argv[2];
var parsedUrl = url.parse(argUrl, true);

var options = {
  host: null,
  port: -1,
  path: null,
  method: 'GET'
};

options.host = parsedUrl.hostname;
options.port = parsedUrl.port;
options.path = parsedUrl.pathname;
if (parsedUrl.search) options.path += "?"+parsedUrl.search;

var req = http.request(options, function(res) {
  util.log('STATUS: ' + res.statusCode);
  util.log('HEADERS: ' + util.inspect(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    util.log('BODY: ' + chunk);
  });
  res.on('error', function(err) {
    util.log('RESPONSE ERROR: ' + err);
  });
});
req.on('error', function(err) {
  util.log('REQUEST ERROR: ' + err);
});
req.end();
```

按如下方式运行这一脚本：

```
$ node wget.js http://example.com
11 Apr 21:34:35 - STATUS: 302
11 Apr 21:34:35 - HEADERS:{"location":"http://www.iana.org/domains/example/",
"server":"BigIP","connection":"close","content-length":"0"}
```

例子中展示了一个状态码为302（重定向）的HTTP响应，它告诉浏览器需要跳转到http://www.iana.org/domains/example/。事实上，如果在浏览器中访问http://example.com，最后会跳转到iana.org。

`wget.js`的作用就是发送HTTP请求，然后显示响应的各种细节。

HTTP请求在`http.request`方法中被初始化，如下：

```
var http = request('http');

var options = {
  host: 'example.com',
  port: 80,
  path: null,
  method: 'GET'
};
var request = http.request(options,
  function(response) { .. });
```

`options`对象描述了将要发出的请求，得到响应后其中的`callback`函数会执行。`options`对象通过`host`、`port`和`path`直接指定了请求的URL。请求的`method`字段必须是一个HTTP动作（GET、PUT、POST等）。我们也可以在一个`headers`数组中存放HTTP请求的头信息。例如，提供一个cookie:

```
var options={

  headers: {
    'Cookie': '.. cookie value'
  }
};
```

`response`对象是一个EventEmitter，它能派发`data`和`error`事件。`data`事件在数据到达时被触发，`error`事件在发生错误时被触发。

`request`对象是一种WritableStream，它在HTTP请求中携带数据（例如PUT或POST）时很有用。HTTP请求中的数据格式通过MIME协议来声明。例如，提交HTML表单时它的Content-Type会被设置成`multipart/form-data`。

要在HTTP客户端请求中发送数据，我们只需调用`.write`方法并写入符合规范的数据，HTTP协议中定义了许多用途广泛的选项来声明数据格式。本书不会精确列出各种HTTP请求的格式，你可以通过以下代码库来研究它们：

- https://github.com/coolaj86/abstract-http-request——HTTP请求系统的高级包装器；
- https://github.com/danwrong/restler——REST客户端库；
- https://github.com/maxpert/Reston——REST客户端库；
- https://github.com/pfleidi/node-wwwdude——REST客户端库；
- https://github.com/cloudhead/http-console——HTTP请求的交互Shell。

5.8　小结

本章我们学习了如下知识：

- EventEmitter以及它们在HTTP客户端和服务器对象中所扮演的角色；
- 使用EventEmitter来分离HTTP数据请求过程和数据接收机制；

- 为了辅助调试，监听HTTP对象或其他EventEmitter的所有事件；
- 实现HTTP服务器；
- 在HTTP服务器中路由接收到的请求；
- 使用MIME协议标识内容的数据类型；
- 实现HTTP客户端。

目前为止，我们已经学习了使用Node实现一个Web应用的基本知识，下一个目标是开发更有用的应用程序。这意味着我们要存储和操作数据。下一章，我们将讨论在外部数据存储空间中存取数据的几种方式。

第 6 章

存取数据

为了给本书画上一个圆满的句号，我们再看看使用Node存取数据的方法。作为一个Web框架，Express无论有多么强大，要是没有数据存储服务的支持，它的用途始终有限。通常我们的最佳做法是使用数据库存储数据。从传统的SQL数据库，到现代NoSQL中面向文档的数据库、键值对数据存储，还有像YQL这样提供基础查询服务的在线数据库，种类繁多的数据库技术能够满足各种各样的应用场景。

在这一章中，我们将实现两个不同的便签应用程序。这两个应用程序基于Express框架构建，展示了CRUD（创建、读取、更新和删除）操作，还用到了Node的SQL和MongoDB模块。

6.1　Node 的数据存储引擎

除了在文件系统中读写文件的功能，Node未能原生支持任何数据存储功能。使用类似数据库等其他数据存储系统则意味着我们需要使用数据库交互模块。Node的wiki列出了一些数据库交互模块，它们涉及与CouchDB、MongoDB、MySQL、Postgres、SQLite3、Memcache、REDIS、YQL等的交互。

详细内容可以参考https://github.com/joyent/node/wiki/modules#database。

一般情况下，安装模块的同时还需要安装模块依赖的其他部分，包括数据库客户端。例如，MySQL模块依赖一个MySQL服务器和一个MySQL客户端库。

6.2　SQLite3——轻量级的进程内 SQL 引擎

SQL数据库并不必然依赖重量级的数据库服务器和高薪数据库管理人员。SQLite3（http://www.sqlite.org/）的安装非常方便，它是一个无服务器且无需配置的SQL数据库引擎，仅仅是作为一个独立的库被链接到应用程序上。node-sqlite3（https:// github.com/developmentseed/node-sqlite3）就是sqlite3的Node版本。

6.2.1　安装 SQLite 3

如果已经安装了npm，SQLite 3的安装就非常简单：

```
$ npm install sqlite3
```

安装此模块之前需要先在系统上安装sqlite3库，还有包含了原生代码（基于C语言）用于链接sqlite3的npm模块。Mac OS X已经安装了sqlite3，如果你使用的Linux版本还没有安装sqlite3，安装过程也只是执行包管理器的一行命令（例如apt-get install libsqlite3）而已。使用这一数据库及其命令行工具的方法和C语言的API都可以在sqlite3的Web站点（http://sqlite.org/）上找到。

6.2.2　用 SQLite3 实现便签应用

为了探索sqlite3的用法，我们将实现一个简单的应用程序，它可以输入和显示便签。随后，这个应用程序还会用MongoDB实现一遍。

由于这个应用是基于SQL数据库构建的，所以Notes数据库的结构需要用SQL语言描述，CREATE TABLE这个SQL命令存在于notesdb-sqlite3.js的下述代码中：

```
CREATE TABLE IF NOT EXISTS notes (
   ts DATETIME,
   author VARCHAR(255),
   note TEXT
)
```

ts字段是一个时间戳，用于标识便签，author字段用于标识便签作者，note字段则包含便签的完整内容。

1. 数据库抽象模块——nodesdb-sqlite3.js

这个模块是一个数据库接口库，它把SQL命令封装在一个模块中，然后提供给应用程序的其他部分使用。它通过函数add（添加便签）、findNoteById（查找并读取便签）、edit（编辑便签）和delete（删除便签）为便签应用全面实现了CRUD操作功能。

这个模块的目的是封装对SQLite3的调用。它提供一些方法用于建立数据库表、在表中插入数据项、返回表中的所有数据行，还有删除表中的数据项从而帮助我们实现MVC架构。

```
var util    = require('util');
var sqlite3 = require('sqlite3');
sqlite3.verbose();
var db = undefined;
exports.connect = function(callback) {
  db = new sqlite3.Database("chap06.sqlite3",
    sqlite3.OPEN_READWRITE | sqlite3.OPEN_CREATE,
    function(err) {
      if (err) {
        utils.log('FAIL on creating database ' + err);
        callback(err);
      } else
        callback(null)
    }
  );
}
exports.disconnect = function(callback) {
  callback(null);
}
```

```
exports.setup = function(callback) {
  db.run("CREATE TABLE IF NOT EXISTS notes "+
    "(ts DATETIME, author VARCHAR(255), note TEXT)",
    function(err) {
      if (err) {
        util.log('FAIL on creating table ' + err);
        callback(error);
      } else
        callback(null);
    });
}
```

以上是实现管理功能的代码，通过模块和函数提供打开、关闭和设置数据库表的功能。数据库名是直接硬编码的，所以当调用connect和setup函数时，当前目录中就会生成chap06.sqlite文件。

本章的后面部分会介绍一个名为notesdb-mongoose.js的模块，它和此处的模块共享同一个API。notesdb-mongoose使用Mongoose模块与MongoDB实例通信，例如此处disconnect函数是空的，但是后面的Mongoose版本会真正断开与Mongoose的连接。

```
exports.emptyNote = { "ts": "", author: "", note: "" };
exports.add = function(author, note, callback) {
  db.run("INSERT INTO notes ( ts, author, note) "+
    "VALUES ( ?, ? , ? );",
    [ new Date(), author, note ],
    function(error) {
      if (error) {
        util.log('FAIL to add ' + error);
        callback(error);
      } else
        callback(null);
    });
}
```

add函数直接使用SQL语句在数据库中添加数据项。

SQLite3对应的.run函数接受一个字符串参数，其中?表示占位符，如例子所示，占位符的值必须通过一个数组传递进来。这种使用字符串参数的方式普遍存在于各种语言的SQL实现中。对于SQLite3，你需要传递一个数组，数组中的每一项对应SQL语句中的一个“?”符号。然后SQL接口会自动将其转换成SQL语句。

值得注意的是调用者提供了一个回调函数，然后通过这个回调函数来声明错误。模型本身不知道如何向用户展示错误，所以只能期望调用函数更好地处理错误了。

```
exports.delete = function(ts, callback) {
  db.run("DELETE FROM notes WHERE ts = ?;",
    [ ts ],
    function(err) {
      if (err) {
        util.log('FAIL to delete ' + err);
        callback(err);
      } else
        callback(null);
```

```
    });
}
```

这里的delete函数负责从数据库中删除便签。

需要注意的是，我们使用时间戳标识需要删除的便签，且整个模块中我们都使用时间戳标记待操作的标签。数据库中的ts字段在add函数中被初始化：

```
exports.edit = function(ts, author, note, callback) {
  db.run("UPDATE notes "+
    "SET ts = ?, author = ?, note = ? "+
    "WHERE ts = ?",
    [ ts, author, note, ts ],
    function(err) {
      if (err) {
        util.log('FAIL on updating table ' + err);
        callback(err);
      } else
        callback(null);
    });
}
```

edit函数的作用是更新便签。例子中使用SQL语句UPDATE，以新的便签内容和时间戳作为参数来更新便签：

```
exports.allNotes = function(callback) {
  util.log(' in allNote');
  db.all("SELECT * FROM notes", callback);
}
exports.forAll = function(doEach, done) {
  db.each("SELECT * FROM notes", function(err, row) {
    if (err) {
      util.log('FAIL to retrieve row ' + err);
      done(err, null);
    } else {
      doEach(null, row);
    }
  }, done);
}
```

allNotes和forAll函数是操作所有便签条目的两种方法。allNotes把数据库中所有的数据行收集到一个数组里，而forAll方法可以接受两个回调函数：每当从数据集中拿到一行数据，回调函数doEach都会执行一遍；当读完所有数据时，回调函数done就会执行。

forAll每次只处理一行数据，相比而言，很显然allNotes会占用更大的内存。

```
exports.findNoteById = function(ts, callback) {
  var didOne = false;
  db.each("SELECT * FROM notes WHERE ts = ?",
    [ ts ],
  function(err, row) {
    if (err) {
      util.log('FAIL to retrieve row ' + err);
      callback(err, null);
    } else {
```

```
        if (!didOne) {
          callback(null, row);
          didOne = true;
        }
      }
    });
  }
```

findNotesById函数返回以时间戳标识的一个便签的相关数据，一个时间戳能标识数据库中的一些特定行。我们还使用了一个标志来保证回调函数只被执行一次，以免处理其他具有相同时间戳的数据行。

2. 初始化数据库——setup.js

因为Node的sqlite3模块基于sqlite3库运作，所以一般的sqlite3工具都能处理用note-sqlite3创建的数据库。例如，我们可以使用如下的sqlite3命令创建一个数据库。

```
Terminal — bash — 88×9
$ sqlite3 chap06.sqlite3
SQLite version 3.7.6
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> CREATE TABLE notes (ts DATETIME, author VARCHAR(255), note TEXT);
sqlite> .schema notes
CREATE TABLE notes (ts DATETIME, author VARCHAR(255), note TEXT);
sqlite> 
```

我们也可以借助notesdb模块，写一个setup.js脚本来初始化数据库：

```
var util = require('util');
var async = require('async');
var notesdb = require('./notesdb-sqlite3');
// var notesdb = require('./notesdb-mongoose');
notesdb.connect(function(error) {
  if (error) throw error;
});
notesdb.setup(function(error) {
  if (error) {
    util.log('ERROR ' + error);
    throw error;
  }
  async.series([
    function(cb) {
      notesdb.add("Lorem Ipsum ",
        "Cras metus. Sed aliquet risus a tortor. Integer id quam.
        Morbi .. fermentum non, convallis id, sagittis at, neque.",
      function(error) {
        if (error) util.log('ERROR ' + error);
        cb(error);
      });
    }
  ],
    function(error, results) {
      if (error) util.log('ERROR ' + error);
      notesdb.disconnect(function(err) { });
```

```
      }
    );
  });
```

值得注意的一点是两次请求调用不同的notesdb模块，然而真正执行的只有require('notesdb-sqlite3')。之后我们会在Mongoose模块中重用这一脚本，由于内部的API是一样的，我们只需要通过修改模块名完成数据库的切换。

之前我们已经对数据库进行了填充，你还可以按照自己的喜好重复notesdb.add操作。我们在这里需要考虑的问题是在什么时候调用.disconnect函数。如果我们在所有的add操作完成前调用了disconnect方法，一部分add操作可能会执行失败。注意，这些函数是异步执行的，add操作的执行顺序可能与其在源代码中定义的不一致。

在这里async模块用于在调用disconnect方法后正确调控一些add操作的执行。正常情况下回调函数会在后台运行，如果在notesdb.disconnect方法后脚本调用了好几次notesdb.add，disconnect操作可能会在所有add操作完成之前运行。async模块是非常重要的。它可以完成很多工作，async.series函数可以控制函数按顺序执行，从而保证最后的函数在所有其他函数完成之后执行。

3. 在控制台显示便签——show.js

如同先前提到的，notesdb.forAll方法能检索到数据库中的每条便签，所以我们能利用它将数据库信息输出到控制台，代码如下：

```
var util = require('util');
var notesdb = require('./notesdb-sqlite3');
// var notesdb = require('./notesdb-mongoose');
notesdb.connect(function(error) {
  if (error) throw error;
});
notesdb.forAll(function(error, row) {
  util.log('ROW: ' + util.inspect(row));
}, function(error) {
  if (error) throw error;
  util.log('ALL DONE');
  notesdb.disconnect(function(err) { });
});
```

我们可以按如下方式运行脚本。

```
Terminal — bash — 90×15
$ node show
6 Jul 15:39:32 - ROW: { ts: 1309991225472,
  author: 'Lorem Ipsum 12',
  note: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec odio. Praese
nt libero. Sed cursus ante dapibus diam. Sed nisi. Nulla quis sem at nibh elementum imperd
iet. Duis sagittis ipsum. Praesent mauris. Fusce nec tellus sed augue semper porta. Mauris
 massa. Vestibulum lacinia arcu eget nulla. Class aptent taciti sociosqu ad litora torquen
t per conubia nostra, per inceptos himenaeos. Curabitur sodales ligula in libero. Sed dign
issim lacinia nunc.' }
6 Jul 15:39:32 - ROW: { ts: 1309991225490,
  author: 'Lorem Ipsum 23',
  note: 'Curabitur tortor. Pellentesque nibh. Aenean quam. In scelerisque sem at dolor. Ma
ecenas mattis. Sed convallis tristique sem. Proin ut ligula vel nunc egestas porttitor. Mo
rbi lectus risus, iaculis vel, suscipit quis, luctus non, massa. Fusce ac turpis quis ligu
la lacinia aliquet. Mauris ipsum. Nulla metus metus, ullamcorper vel, tincidunt sed, euism
```

4.整合便签应用——app.js

现在我们已经知道如何通过notesdb-sqlite3.js操作数据库，让我们将其整合到一个基于Express的简单应用中，其中notesdb-sqlite3.js作为便签应用的模型部分，而app.js会扮演控制器的角色。我们马上会看到一些模板文件，这些文件将帮助展示这一方法。和我们已经看过的两个脚本文件show.js和setup.js一样，app.js也参照了之前的写法，可以轻易在notesdb-sqlite3.js和notesdb-mongoose.js模块之间切换：

```
var util    = require('util');
var url     = require('url');
var express = require('express');
var nmDbEngine = 'sqlite3';
// var nmDbEngine = 'mongoose';
var notesdb = require('./notesdb-'+nmDbEngine);
var app = express.createServer();
app.use(express.logger());
app.use(express.bodyParser());
app.register('.html', require('ejs'));
app.set('views', __dirname + '/views-'+nmDbEngine);
app.set('view engine', 'ejs');
```

这部分代码用于加载必需的模块，并配置Express服务器组件。

这里我们需要重点介绍nmDbEngine变量和它的使用。这个变量用于命名数据库引擎、选择合适的notesdb实现和选择合适的views目录。它们都因数据库引擎的不同而不同，不过app.js可以保持不变：

```
var parseUrlParams = function(req, res, next) {
  req.urlP = url.parse(req.url, true);
  next();
}
notesdb.connect(function(error) {
  if (error) throw error;
});
app.on('close', function(errno) {
  notesdb.disconnect(function(err) { });
});
```

在此，connect和disconnect函数用于维护数据库的连接。

parseUrlParams函数是一个路由中间件函数，用于在一些路由器函数中解析URL查询参数：

```
app.get('/', function(req, res) { res.redirect('/view'); });
app.get('/view', function(req, res) {
  notesdb.allNotes(function(err, notes) {
    if (err) {
      util.log('ERROR ' + err);
      throw err;
    } else
      res.render('viewnotes.html', {
        title: "Notes ("+nmDbEngine+")", notes: notes
      });
  });
});
```

这里我们在浏览器中展示了一系列便签。

首先我们要做的是通过调用res.redirect('/view')将一个请求从/重定向到/view。/view页面会被作为便签应用的主界面，许多路由器函数都会将请求重定向到这个页面。

页面的渲染由viewnotes.html模板完成，我们会在后面介绍这部分内容。渲染过程调用了两个变量，title用于包含页面标题字符串，notes是一个便签数组。该模板会处理并渲染所给出数组中所有的便签：

```
app.get('/add', function(req, res) {
  res.render('addedit.html', {
    title: "Notes ("+nmDbEngine+")",
      postpath: '/add',
      note: notesdb.emptyNote
  });
});
app.post('/add', function(req, res) {
  notesdb.add(req.body.author, req.body.note,
    function(error) {
      if (error) throw error;
      res.redirect('/view');
    });
});
```

我们在此通过一些路由函数添加便签到数据库中。

其中需要讨论的实现细节是两个负责路由到/add目录的函数。当用户单击Add按钮时app.get('/add',...)内的函数就会被调用。浏览器会发出一个发往/add的HTTP GET请求。这个函数使用addedit.html模板来创建一个表单，让用户通过这个表单输入标签，然后通过单击Submit按钮提交。

模板addedit.html用于处理/add和/edit操作，它期望接受一个Note对象。Notesdb.emptyNote对象是一个空的便签，适合在不存在便签对象时使用。

在用户提交表单时，浏览器就会发起一个HTTP POST请求，app.post('/add',...)中的函数就会被调用。用户输入的数据会被存放在请求主体中，而请求主体会被bodyParser(app.use(express.bodyParser()))中间件处理并存放在req.body中。最后我们可以通过req.body.author和req.body.note使用用户输入的数据。

```
app.get('/del', parseUrlParams, function(req, res) {
  notesdb.delete(req.urlP.query.id,
    function(error) {
      if (error) throw error;
      res.redirect('/view');
    });
});
```

这个路由器函数用于删除数据库中的便签。

我们将parseUrlParmas用作路由中间件的理由是标签标识符会作为URL查询参数出现，并且标识符以id命名。因此，我们可以继续并通过编写req.urlP.query.id访问id参数，而不用再次编写解析URL的代码。当notesdb.delete操作完成后，我们会将应用重定向到/view处的主页。

```
app.get('/edit', parseUrlParams, function(req, res) {
  notesdb.findNoteById(req.urlP.query.id,
    function(error, note) {
      if (error) throw error;
      res.render('addedit.html', {
        title: "Notes ("+nmDbEngine+")",
        postpath: '/edit',
        note: note
      });
    });
});
app.post('/edit', function(req, res) {
  notesdb.edit(req.body.id, req.body.author, req.body.note,
    function(error) {
      if (error) throw error;
      res.redirect('/view');
    });
});
app.listen(3000);
```

这个路由器函数用于编辑数据库中的便签。

我们再次使用了parseUrlParams中间件从URL查询参数中获取便签id，并使用这个id和notesdb.findNoteById函数获取便签。注意，我们再次使用addedit.html渲染页面，但是这次发送到页面上的便签取自数据库。

之前postpath变量被设置成/add，而现在要设置成/edit。这个变量是addedit.html中表单提交的目的地，确保调用正确的app.post函数，即调用app.post('/add',...)或者app.post('edit',...)。

5. 便签应用的模板

在运行便签应用之前，我们必须设置在app.js中引用的模板，这些模板包括viewnotes.html、addedit.html和layout.html。

后续的文件必须放置在views-sqlite3目录下。之后我们会创建另外一个目录——views-mongoose，用于存放Mongoose模板。

让我们从layout.html模板开始：

```
<html>
  <head><title><%= title %></title></head>
  <body>
    <h1><%= title %></h1>
    <p><a href='/view'>View</a> | <a href='/add'>Add</a></p>
    <%- body %>
  </body>
<html>
```

这是便签应用的页面布局，非常简洁。它会处理app.js中调用res.render时接收的title变量。

现在看下viewnotes.html模板：

```
<table><% notes.forEach(function(note) { %>
```

```
    <tr><td>
      <p><%= new Date(note.ts).toString() %>:
        by <b><%= note.author %></b></p>
      <p><%= note.note %></p>
    </td><td>
      <form method='GET' action='/del'>
        <input type='submit' value='Delete' />
        <input type='hidden' name='id' value='<%=
          note.ts %>'>
      </form>
      <br/><form method='GET' action='/edit'>
        <input type='submit' value='Edit' />
        <input type='hidden' name='id' value='<%=
          note.ts %>'>
      </form>
    </td></tr><% }); %></table>
```

这个版本适用于基于SQLite3的便签应用。它展示了标签的时间戳、标题和内容，还有两个让用户删除/del或者编辑/edit便签的表单。

这里有一个隐藏的表单值id，而在基于SQLite3的标签应用中我们使用时间戳来标识便签。之前针对删除或编辑操作讨论app.post函数时，我们通过解析URL参数来获取id参数，而这个id就来自表单中的隐藏值。

现在看下addedit.html：

```
<form method='POST' action='<%= postpath %>'>
  <% if (note) { %>
    <input type='hidden' name='id' value='<%= note.ts %>'>
    <% } %>
    <input type='text' name='author' value='<%= note.author %>'/>
    <br/>
    <textarea rows=5 cols=40 name='note' ><%=
      note.note
    %></textarea>
    <br/><input type='submit' value='Submit' />
</form>
```

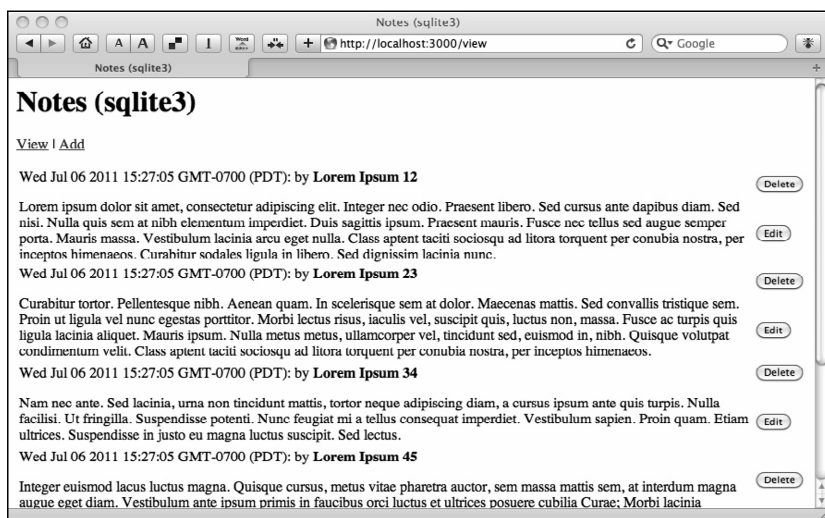
这个表单在添加（/add）和编辑（/edit）便签时都会用到。postpath变量的值被设定为表单的目的URL。其他表单值从app.js传递来的便签对象中获取，这里有可能是emptyNote对象。

6. 启动基于SQLite3的便签应用

现在我们已经把各个部分整合成一个应用，便签应用已经能够正常运行。如果你在之前已经运行过setup.js，那么数据库也已经配置好了，如果还没有运行过，那现在运行一下。我们现在可以通过下面的方式运行应用。

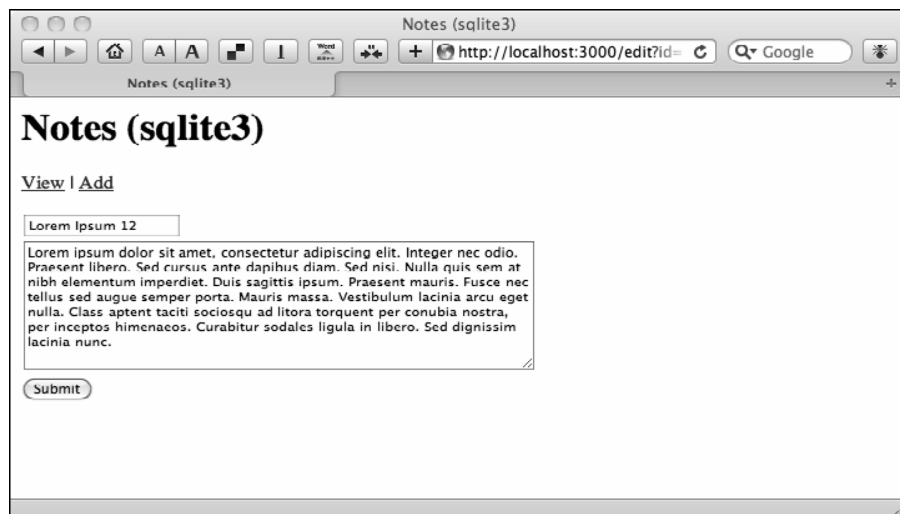
```
$ node app
```

由于执行了app.listen(3000)语句，所以我们可以在http://localhost:3000/上访问应用。大致的界面如下。



如果你单击了Delete按钮，浏览器会快速刷新页面，然后你会发现已单击删除的条目已经消失。因为`app. get('/del'..)`只是调用了`notesdb.delete`方法，然后马上重定向到`/view`，所以会有一个快速的刷新过程。

接下来我们可以添加（单击Add链接）或者编辑（单击Edit按钮）便签，如图所示。



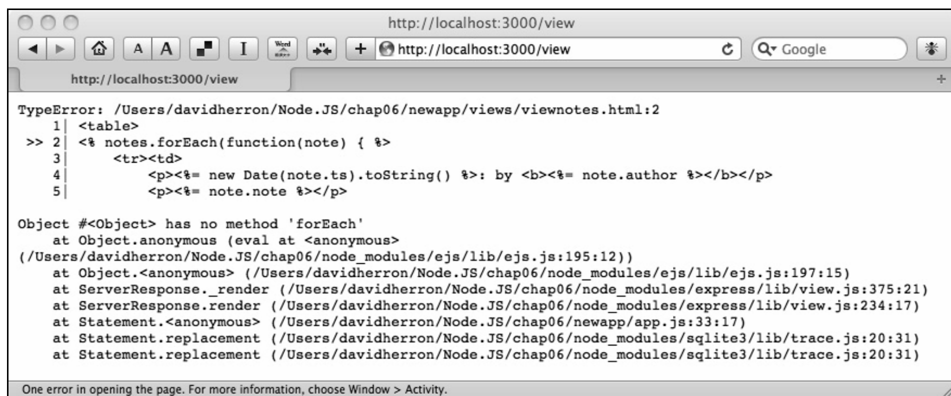
单击Submit按钮后，应用会调用`app.post('/add',..)`或者`app.post('/ edit',..)`，两者都会更新数据库然后将页面重定向到`/view`。

7. 处理和调试错误

如果你的代码有错误或者有其他问题发生，应用内部会抛出错误对象。在应用中调试错误意味着要将错误展示出来，以便了解错误发生在何时何处。在便签应用中我们使用`util.log`语句

实现错误的显示。在`notesdb-sqlite3.js`模块中，我们使用回调函数发送错误对象到`app.js`，然后`app.js`抛出错误。

Express中默认含有的一个重要部分，就是在浏览器中展示对开发者友好的栈轨迹。①



Express提供的`app.error`函数用于捕获`route`函数抛出的异常，或者在调用`next(error)`时执行。

为了探索错误机制，我们可以简单地制造一个错误，比如在一个`null`对象中调用一个方法：

```
app.get('/del', parseUrlParams, function(req, res) {
  var notAllowed = null;
  notAllowed.delete();
..
});
```

我们刚刚展示的错误页面对于开发者来说有些用处，但是对用户不够友好，所以我们要对它做一些改进。

一种改进方法是将下面这条命令插入到`app.js`中：

```
app.use(express.errorHandler({ dumpExceptions: true }));
```

浏览器窗口会显示一个简单的消息——Internal Server Error（内部服务器错误）。这对用户来说不会太不友好，但是依然不合适。对开发者友好的栈轨迹确实可以输出到`stderr`，不会因不必要的细节内容影响用户，同时对于开发者来说非常实用。

建立一个合适的对用户友好的页面，其起点是如下的`app.error`函数：

```
app.error(function(err, req, res) {
  res.render('500.html', {
    title: "Notes ("+nmDbEngine+") ERROR", error: err
  });
});
```

实现这个函数的方式有很多，比如基于接收的错误对象的类型生成不同的错误页面，或者展

① 参见http://en.wikipedia.org/wiki/Stack_trace。

示一张小鸟从海中叼起一条鲸鱼的图片。具体情况可以你自己决定，出于演示的目的，我们使用这个错误页面模版500.html：

```
<b>Internal Server Error</b>
ERROR: <%= error %>
```

这些都就绪后，我们就将这个错误展示到了浏览器中。



6.2.3 在 Node 中使用其他 SQL 数据库

SQLites只是众多数据库中的一个。我们选择它来构建便签应用的原因是，它的安装和配置非常方便。在数据库需要放置在一台计算机上时，你应该考虑使用SQLite3。其他SQL数据库有另外一些引人注目的特性，比如支持分布式数据库访问、较高的吞吐量、备份功能等。

底层数据库（接近SQL）如下所示。

- ❑ Node-mysql（https://github.com/fleixge/node-mysql）是一个纯粹通过JavaScript实现MySQL客户端协议的数据库。
- ❑ Node-mysql-native（https://github.com/sidorares/nodejs-mysql-native）将原生的MySQL客户端库包装成一个Node模块。
- ❑ Node-mysql-libmysqlclient(https://github.com/Sannis/node-mysql-libmysqlclient)作为MySQL的插件为Node提供MySQL客户端服务。
- ❑ Node-postgres（https://github.com/brianc/node-postgres）是一个通过严格测试的Node客户端，用于连接Postgres。它同时含有JavaScript版本和原生插件。
- ❑ Node-sqlite3（https://github.com/developmentseed/node-sqlite3）是一个为Node设计的异步非阻塞SQLite3插件。
- ❑ Node-DBI（https://github.com/DrBenton/Node-DBI） 是一个SQL数据库抽象层。

高级数据库（含有ORM特性）如下所示。

- ❑ FastLegS（https://github.com/didit-tech/FastLegS）是一种PostgreSQL ORM，基于node-postgres构建。
- ❑ Node-orm（https://github.com/dresende/node-orm）是Node中对象关系映射器，适用于多重数据库。
- ❑ persistence.js（https://github.com/zefhemel/persistencejs）是一个异步JavaScript对象关系映

射库，可以同时在浏览器和Node服务器应用中使用。

- Sequelize（https://github.com/sdepold/sequelize）是一个用于Node和MySQL之间的对象关系映射器。

6.3　Mongoose

MongoDB是“nosql”数据库的领头羊之一。（nosql当然也意味着不用SQL。）它被誉为“可扩展、高性能、开源、面向文档的数据库”。它使用JSON风格的文档，不用事先定义的严格模式，拥有大量先进的特性。你可以在其网站上看到更多文档和详细内容，网址为www.mongodb. org/。

Mongoose是用于访问MongoDB的模块之一。它是一个对象建模工具，意味着你的程序负责定义模式对象来描述数据，而Mongoose负责数据到MongoDB的存储。Mongoose对于Node和MongoDb而言是一个非常强大的对象建模工具，使用嵌入式文档，是一个类型灵活的系统，适用于字段输入、字段验证、虚拟字段等，详见http://mongoosejs.com/。

6.3.1　安装 Mongoose

如果你已经安装了npm，安装Mongoose就很容易了：

```
$ npm install mongoose
```

在使用Mongoose之前，你需要运行一个MongoDB实例。在mongodb.com上有预建的二进制包，你也可以通过大多数Linux系统上的包管理系统安装。在Mac OS X上你也可以通过MacPorts安装。你可以访问它们的网站获取更多信息，特别是针对具体操作系统的快速入门向导（www. mongodb.org/display/DOCS/Quickstart）。

我们需要两个步骤来验证是否已经可以使用MongoDB。第一步是在本地数据目录启动MongoDB服务器（`mongod`），如下面的截图所示。

```
Terminal — mongod — 90×11
$
$ rm -rf data
$ mkdir data
$ mongod --dbpath ./data
Thu Jul  7 15:19:22 MongoDB starting : pid=18682 port=27017 dbpath=./data 64-bit
Thu Jul  7 15:19:22 db version v1.6.6-pre-, pdfile version 4.5
Thu Jul  7 15:19:22 git version: nogitversion
Thu Jul  7 15:19:22 sys info: Darwin tippy.local 10.7.0 Darwin Kernel Version 10.7.0: Sat
Jan 29 15:17:16 PST 2011; root:xnu-1504.9.37~1/RELEASE_I386 i386 BOOST_LIB_VERSION=1_45
Thu Jul  7 15:19:22 [initandlisten] waiting for connections on port 27017
Thu Jul  7 15:19:22 [websvr] web admin interface listening on port 28017
```

在开发的时候这个方法非常实用，并且不需要耗费太多时间。你可以在任何时候用*Control+C*结束进程，并用这里的命令在一个干净的目录下重新开始。

下一步是在快速入门向导（在前面的链接中可以查看）中通过用户交互检查是否可使用MongoDB。

```
Terminal — mongo — 72×8
$ mongo
MongoDB shell version: 1.6.6-pre-
connecting to: test
> db.foo.save( { a : 1 } )
> db.foo.find()
{ "_id" : ObjectId("4e16313464a1126e2a76a030"), "a" : 1 }
>
```

例子里的操作会插入一个文档（JSON格式的`{a: 1}`）到名为`foo`的集合中。命令`db.foo.find`用于查询集合`foo`并以JSON格式列出集合中的所有元素（因为没有查询参数）。MongoDB网站上有完整的使用文档，包括Mongo shell的使用方法。

6.3.2 用 Mongoose 实现便签应用

为了学习Mongoose的使用方法，我们将实现便签应用的另外一个版本。

构架基本和SQL版本类似，不过编写的时候使用的是Mongoose对象标识符。

```
var NoteSchema = new Schema({
  ts     : { type: Date, default: Date.now },
  author : String,
  note   : String
});
mongoose.model('Note', NoteSchema);
```

此处使用字段的目的也和使用SQL时一致。因为Mongoose使用JavaScript实现，所以我们使用的数据类型也是JavaScript对象。以防在创建对象的时候没有提供`ts`值，此处为其使用默认值。

现在开始编码了。

1. 数据库抽象模块——notesdb-mongoose.js

和`sqlite3`便签应用一样，这个模块被作为数据库接口库被应用的其他部分使用。它实现了3种数据库CRUD功能，具体表现为`add`（创建）、`findNoteById`（读取）、`edit`（更新）和`delete`（删除）这3个标签文档操作函数。

与基于`sqlite3`的便签应用一样，`notesdb-mongoose.js`实现了MVC架构中的模型部分：

```
var util = require('util');
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var dburl = 'mongodb://localhost/chap06';
exports.connect = function(callback) {
  mongoose.connect(dburl);
}
exports.disconnect = function(callback) {
  mongoose.disconnect(callback);
}
```

这段管理代码引入了一个模块并添加了`.connect`和`.disconnect`函数。变量`dburl`用于连接已运行的MongoDB。这些合在一起负责处理与MongoDB的连接，而程序本身只需要在启动的时候调用`.connect`，并停止前调用`.disconnect`。

```
exports.setup = function(callback) { callback(null); }
var NoteSchema = new Schema({
  ts     : { type: Date, default: Date.now },
  author : String,
  note   : String
});
mongoose.model('Note', NoteSchema);
var Note = mongoose.model('Note');
exports.emptyNote = { "_id": "", author: "", note: "" };
```

这部分代码负责定义模式，不过我们已经在前面讨论过了。这一模式的创建方法是`var NoteSchema = new Schema(...)`。我们用如下代码将其作为Mongoose的模型注册进去：

```
mongoose.model('Note', NoteSchema);
var Note = mongoose.model('Note');
```

在注册完模式和模型后，你的程序就可以在数据库中创建文档了：

```
exports.add = function(author, note, callback) {
  var newNote = new Note();
  newNote.author = author;
  newNote.note   = note;
  newNote.save(function(err) {
    if (err) {
      util.log('FATAL '+ err);
      callback(err);
    } else
      callback(null);
  });
}
```

我们通过创建对象的一个新实例，将数据赋给相应字段，并调用`.save`方法。在这个例子里，我们没有给`ts`字段赋值，不过模式定义会声明一个默认值：

```
exports.delete = function(id, callback) {
exports.findNoteById(id, function(err, doc) {
    if (err)
      callback(err);
    else {
      util.log(util.inspect(doc));
      doc.remove();
      callback(null);
    }
  });
}
```

从数据库删除一条便签需要两个步骤。我们需要先使用`findNoteById`方法检索便签，然后调用对象的`.remove`方法。我们会在稍后进行展示。

```
exports.edit = function(id, author, note, callback) {
  exports.findNoteById(id, function(err, doc) {
    if (err)
     callback(err);
    else {
      doc.ts     = new Date();
```

```
        doc.author = author;
        doc.note   = note;
        doc.save(function(err) {
          if (err) {
            util.log('FATAL '+ err);
            callback(err);
          } else
           callback(null);
        });
      }
    });
  }
```

同样，更新一条便签也需要两个步骤。首先我们要检索便签，然后给其字段赋上新的值，再调用.save方法进行保存：

```
exports.allNotes = function(callback) {
  Note.find({}, callback);
}
exports.forAll = function(doEach, done) {
  Note.find({}, function(err, docs) {
    if (err) {
      util.log('FATAL '+ err);
      done(err, null);
    }
    docs.forEach(function(doc) {
      doEach(null, doc);
    });
    done(null);
  });
}
var findNoteById = exports.findNoteById = function(id,
  callback) {
  Note.findOne({ _id: id }, function(err, doc) {
    if (err) {
      util.log('FATAL '+ err);
      callback(err, null);
    }
    callback(null, doc);
  });
}
```

现在我们看看负责在数据库中检索便签的这3个函数。

在allNotes和forAll函数中，我们使用了Notes.find方法。它和后台使用的Querry对象一样都是Mongoose中强大的一部分。它的作用与SQL SELECT语句中的WHERE类似，但是它更加简洁易读。因为这两个函数中的Querry对象都是空的，所以Mongoose会把便签集合中所有的文档都检索出来。

在.findNoteById函数中，我们调用了Note.findOne方法通过_id字段来找到一个特定的便签。我们的方法是传入一个Query对象{_id: id}来用_id字段匹配id。_id字段是MongoDB提供的绝对唯一的ID，用于标识存储的文档。因为它的作用和基于sqlite3的便签应用中的ts字段一样，所以我们使用_id字段值来标识便签。Mongoose的Query对象还有更多用处，具体可

见mongoosejs.org。

2. 初始化数据库——setup.js

和使用MongoDB时一样，此处亦有两种方式来初始化数据库。你可以使用mongo shell命令，如下面的截图所示。

```
Terminal — mongo — 76×12
$
$ mongo
MongoDB shell version: 1.6.6-pre-
connecting to: test
> use chap06;
switched to db chap06
> db.notes.save({ ts: "Tue May 10 2011 20:26:38 GMT-0700 (PDT)", author: "so
meone", note: "A meaningful note" });
> db.notes.find();
{ "_id" : ObjectId("4e16484f64f7b4392f563b7a"), "ts" : "Tue May 10 2011 20:2
6:38 GMT-0700 (PDT)", "author" : "someone", "note" : "A meaningful note" }
> 
```

另一种方式是使用之前的setup.js脚本。它里面的一对代码行可以实现notesdb-sqlites3和notesdb-mongoose之间的切换。

```
// var notesdb = require('./notesdb-sqlite3');
var notesdb = require('./notesdb-mongoose');
```

通过注释掉不同的语句完成切换后，再按照下面的方式运行脚本：

```
$ node setup
```

脚本本身不会输出任何内容，但是你可以使用show.js显示并查看数据库。

3. 在控制台中显示便签——show.js

我们可使用之前的show.js显示数据库中的每一个条目。我们只需和在前面的setup.js中一样做修改，按照如下的方式运行脚本：

```
$ node show
7 Jul 17:20:58 - ROW: { doc:
  { ts: Fri, 08 Jul 2011 00:13:22 GMT,
    _id: 4e164ba289dc189149000001,
    note: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      Integer nec odio. Praesent .. Sed dignissim lacinia nunc.',
   author: 'Lorem Ipsum 12' },
  activePaths:
  { paths:
    { note: 'init',
      author: 'init',
      _id: 'init',
      ts: 'init' },
    states: { init: [Object], modify: {}, require: {} },
    stateNames: [ 'require', 'modify', 'init' ] },
  saveError: null,
  isNew: false,
  pres: { save: { serial: [Object], parallel: [] } },
  errors: undefined }
```

4. 整合应用 —— app.js

由于`notesdb-mongoose.js`的API和`notesdb-sqlite.js`的一样，我们可以通过极小的修改重用`setup.js`和`show.js`。这对于`app.js`来说也一样。修改本身稍有不同，但意图是相同的。然而，因为有区别存在，所以我们必须使用不同的模板文件。

`app.js`中的修改如下：

```
// var nmDbEngine = 'sqlite3';
var nmDbEngine = 'mongoose';
```

现在，我们需要创建目录`views-mongoose`，然后准备创建下述模板文件。

(1) 首先是`layout.html`，它和之前完全一样，所以只需要复制一份就行:

```
$ cp views-sqlite3/layout.html views-mongoose/layout.html
```

(2) 下一个是`viewnotes.html`，它和之前的一样，不过需要按照下面的方式修改`hidden`类型的`id input`标签：

```
<input type='hidden' name='id' value='<%= note._id %>'>
```

(3) 基本和上面类似，复制`addedit.html`文件，然后按照下面的方式修改`hidden`类型的`id input`标签：

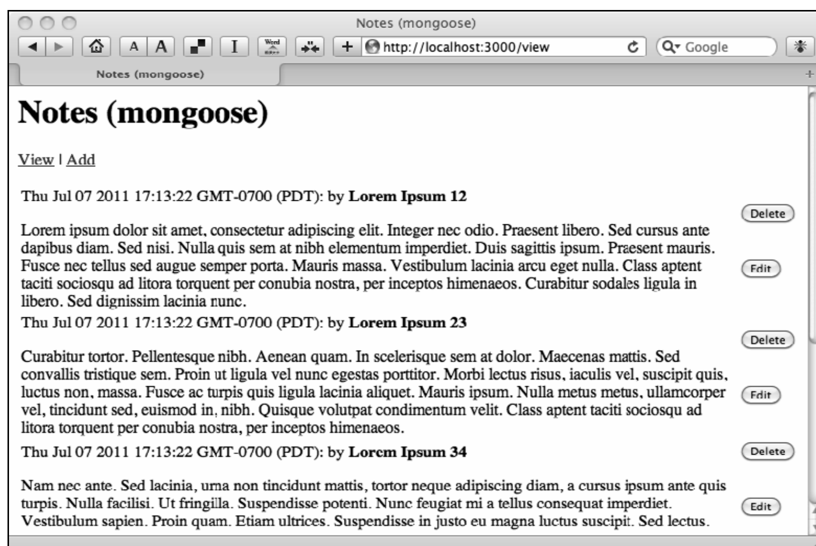
```
<input type='hidden' name='id' value='<%= note._id %>'>
```

这里的模板和SQLite3版的区别是`hidden`类型的表单字段`id`不同。就像我们之前提到的那样，Mongo提供了一个全局唯一的_`id`值来标识每一个已存储的文档。

现在我们已经将应用整合完毕，可以运行Mongoose便签应用了：

```
$ node app
```

然后你可以在浏览器中访问http://localhost:3000/浏览应用。它看起来和之前基于SQLites的版本类似，只是标题会不同，如截图所示。



这个版本的便签应用和之前的运行起来完全一样，两个版本的便签应用分别是用于演示如何用SQL和Mongo数据库构建应用。

6.3.3　对 MongoDB 数据库的其他支持

Mongoose不是唯一一个在Node中使用MongoDB的工具。

你可能会感到惊讶的是MongoDB shell和Node MongoDB模块API的区别。由于MongoDB shell使用了JavaScript命令解释器，你会认为它们的API是一模一样的。尽管许多模块声称和MongoDB shell类似，它们都没有使用一样的API。

- Node-mongodb（https://github.com/orlandov/node-mongodb）是一个面向MongoDB的实验性质的异步Node接口。
- node-mongodb-native（https://github.com/christkv/node-mongodb-native）是另外一个驱动程序。
- node-mongolian（https://github.com/marcello3d/node-mongolian）是一个“了不起”的驱动程序，它试图做到和MongoDB shell非常类似。
- Mongolia（https://github.com/masylum/mongolia）是MongoDB上一个灵活的“非魔法”层，不过不是ORM。
- Mongoose（http://www.learnboost.com/mongoose/）是我们刚刚使用过，基于MongoDB的一个ORM系统。
- Mongous（https://github.com/amark/mongous）是一个非常简单的面向MongoDB的接口，语法和jQuery类似。
- node-nosql-thin（https://github.com/dmcquay/node-nosql-thin）是一个面向MongoDB的非常“瘦小”的接口库，未来可能支持其他“NoSQL数据库”。

6.4　如何实现用户验证

很多应用都需要用户登录，然后用户才能进行一些特权操作。由于HTTP是一个无状态的协议，验证用户的唯一方式就是发送一个cookie到浏览器上，然后验证标识符。cookie包含了应用中用于验证用户的数据。我们将快速了解登录表单的实现、cookie的发送和无cookie时对访问便签应用的阻止。

我们要对`app.js`做两个修改，首先在服务器对象配置中添加`cookieParser`中间件：

```
var app = express.createServer();
app.use(express.logger());
app.use(express.cookieParser());
app.use(express.bodyParser());
```

下一步是添加一个简单的路由中间件函数来检查用户是否被允许访问。在这个例子里，我们只检查cookie是否等于`AOK`，因为这个单词通常意味着一切都没问题。

```
var checkAccess = function(req, res, next) {
```

```
  if (!req.cookies
    || !req.cookies.notesaccess
    || req.cookies.notesaccess !== "AOK") {
      res.redirect('/login');
  } else {
    next();
  }
}
```

cookieParser中间件在这里做了很多工作，查找cookie，解析cookie，然后将解析出来的值放到req对象中。当存在cookie时，它的值会被放入req.cookies中，然后我们就可以如此处所示获取cookie的值。如果没有cookie或者req对象中不存在notesaccess字段，又或者cookie的值不是AOK，浏览器就会重定向到URL为/login的页面。

在了解URL为/login的处理程序之前，我们先添加cookieParser中间件到便签应用路由上。这非常容易：

```
app.get('/view', checkAccess, function(req, res) {
..
});
```

我们特别在每一个路由器函数的定义中调用了checkAccess函数。这可以保证check-Access函数会在每一个便签的URL处理中被调用，也就保证每个便签URL都能受到登录保护。不需要受到登录保护的URL处理也就不需要使用checkAccess路由中间件函数。

这两个路由器函数对应处理URL为/login的页面：

```
app.get('/login', function(req, res) {
  res.render('login.html', {
    title: "Notes LOGIN ("+nmDbEngine+")",
  });
});
app.post('/login', function(req, res) {
  // 检查表单中输入的凭证
  res.cookie('notesaccess', 'AOK');
  res.redirect('/view');
});
```

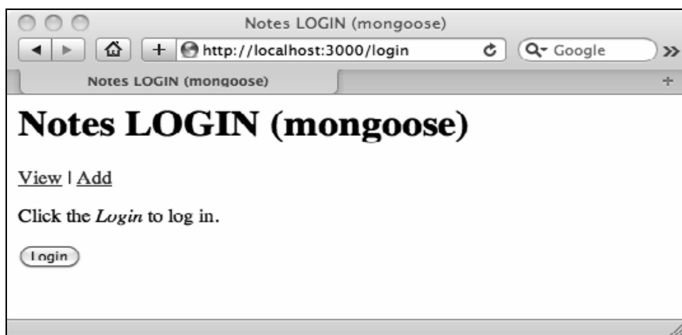
最后使用下面的模板——login.html：

```
<form method='POST' action='/login'>
  <p>Click the <i>Login</i> to log in.</p>
  <input type='submit' value='Login' />
</form>
```

如果你要实现一个真正安全的系统，这里还需要做一些事情。

当checkAccess函数将用户的浏览器重定向到/login页面时，第一个路由器函数会在浏览器中渲染login.html模板，截图如下。

一个真正安全的系统至少需要有输入用户名和密码的输入框。不过我们跳过了这部分，只是让用户单击Login按钮。



按钮本身是表单的一部分，它会引发`app.post('/login'..)`路由函数的执行。如果这是一个真正安全的系统，这个函数会检查登录表单提交上来的用户验证信息，如果数据和用户数据库中的数据匹配则放出一个用于身份验证的cookie。而这里的路由函数值放出了值为`AOK`的cookie，然后将浏览器重定向到`/view`页面。

虽然该系统不是一个完全安全的系统，但是它依然包含了一个安全系统的主干部分。许多网站都含有用户登录表单，对于每一次页面请求使用并验证cookie。我们实现的功能包括检查验证cookie并纠正cookie值、一个到登录表单的重定向、一个登录表单检查，以及向浏览器发送用于身份验证的cookie。

6.5　小结

我们在这一章学习了Node中的许多数据存取知识。数据存取对于各类应用都很关键，下面回顾一下所学的内容。

- Node不包括内置的数据存储引擎支持，但是Node社区已经开发了很多模块来完成与很多现存数据库的交互。
- 安装数据存储引擎模块就是安装服务器和客户端库等依赖。
- SQLite3提供了一个不需要安装和配置的、开发SQL应用的方式。
- 分别基于SQL和MongoDB数据库实现相同的应用。
- 虽然ORM技术最好以SQL数据存储为基础，但是Node社区已经开发了适合MongoDB和CouchDB的ORM。
- 如何（部分）实现MVC构架。
- 在Express应用中处理表单提交。
- 基于文档的数据库系统（如MongoDB）比SQL更接近于现代编程语言和应用。

我们已经在本书中学习了很多知识。我们首先大致了解了Node和可以用Node实现的软件，然后学习了如何分别在开发环境和部署环境下安装Node和npm，并利用这些基础知识开发了Node模块和一些应用，从而学习了如何构建Web应用、HTTP客户端与服务器应用、Node事件循环机制，以及如何将长时间执行的CPU密集型算法转换成依赖Node事件循环机制的多个计算任务、如何使用网络服务分配工作给后台进程，以及如何将数据从数据库转移到Node应用中。

Node Web Development

Node Web开发

作为服务器端的JavaScript解释器，Node是一个轻量高效的开发平台，用于构建响应快速、高度可扩展的Web应用。它使用事件驱动和非阻塞的I/O模型，非常适合开发数据密集、对实时响应要求高的分布式应用，在微软、eBay、LinkedIn、雅虎等世界知名公司均有成功的应用。

本书是Node开发基础教程，通过大量示例介绍如何使用HTTP服务器和客户端对象、Connect和Express应用框架、异步执行算法，以及如何结合使用SQL和MongoDB数据库。另外，本书同时针对开发和部署环境给出了实用的Node安装建议，介绍了HTTP服务器和客户端应用的开发，阐述了很多Node使用方式，包括在应用中使用数据库存储引擎，以及在有无Connect/Express Web应用框架的情况下开发网站的方法。本书还介绍了Node的CommonJS模块系统，帮助开发人员实现一些重要的面向对象设计方案。

本书适合具有一定JavaScript和Web应用开发基础知识、打算使用服务器端JavaScript开发高性能Web应用的开发人员阅读。



- 用Node平台打造高性能Web应用
- 雅虎架构师精准解读最炙手可热的Web开发技术
- 从基础到实践，示例丰富

[PACKT] PUBLISHING

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/网络技术/Node

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-27832-6

定价：35.00元